Leandro Vagner Arcebispo Ferreira

GTK: UMA FERRAMENTA DE AUXÍLIO NA CONSTRUÇÃO DE GRAMÁTICAS LL(1)

Formiga - MG

2018

Leandro Vagner Arcebispo Ferreira

GTK: UMA FERRAMENTA DE AUXÍLIO NA CONSTRUÇÃO DE GRAMÁTICAS LL(1)

Monografia do trabalho de conclusão de curso apresentado ao Instituto Federal Minas Gerais - Campus Formiga, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais

Campus Formiga

Ciência da Computação

Orientador: Prof.º Walace de Almeida Rodrigues

Formiga - MG

2018

Leandro Vagner Arcebispo Ferreira
GTK: UMA FERRAMENTA DE AUXÍLIO NA CONSTRUÇÃO DE GRAMÁTICAS LL(1)/ Leandro Vagner Arcebispo Ferreira. – Formiga - MG, 2018-
81 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof.º Walace de Almeida Rodrigues

Monografia – Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais Campus Formiga
Ciência da Computação, 2018.

1. Compiladores. 2. Gramáticas LL(1). 2. Grammar Tool Kit. I. Walace Rodrigues. II. Instituto Federal de Minas Gerais Campus Formiga. III. Bacharelado em Ciência da Computação. IV. GTK: Uma ferramenta de auxílio na construção de gramáticas LL(1).

Leandro Vagner Arcebispo Ferreira

GTK: UMA FERRAMENTA DE AUXÍLIO NA CONSTRUÇÃO DE GRAMÁTICAS LL(1)

Monografia do trabalho de conclusão de curso apresentado ao Instituto Federal Minas Gerais - Campus Formiga, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Formiga - MG, 21 de novembro de 2018:

Prof.º Walace de Almeida Rodrigues

Prof.º Denise Ferreira Garcia Rezende

Prof.º Diego Mello Da Silva

Formiga - MG
2018

Agradecimentos

Agradeço a minha família, por sempre me permitir escolher meu próprio caminho e proporcionar o apoio necessário para trilhá-lo. E agradeço meus amigos, por caminharem comigo nessa curta, perigosa e incrível estrada chamada vida.

"Saber que o caminho é longo, mas chegar é questão de tempo pra quem gosta de viver."

Autor

Resumo

A proposta deste trabalho é desenvolver um *software* que auxilie os alunos na construção de gramáticas LL(1), que serão utilizadas na contrução de analizadores sintáticos na disciplina de compiladores. Para isto, o presente trabalho propõe a implementação de uma ferramenta que receba a gramática escrita pelo usuário, valide esta entrada e disponibilize os algoritmos presentes na literatura para realizar manipulações nesta gramática. O trabalho desenvolvido tem como objetivo acrescentar na absorção do conteúdo prático nas disciplinas de Compiladores e Linguagens Formais.

Palavras-chave: Compiladores.Gramática Livre de Contexto.LL(1).Analizador Sintático.

Abstract

The purpose of this work is to develop a *software* to assist students in the construction of LL(1) grammars, which will be used in the construction of syntactic analyzers in the Compiler discipline. For this, the present work proposes the implementation of a tool that receives the grammar written by the user, validate this input and make available the algorithms present in the literature to perform manipulations in this grammar. The objective of this work is to add to the absorption of practical content in the Compilers and Formal Languages.

Keywords: Compilers. Context Free .LL (1) Syntax Analyzer.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AD	Árvore de derivação
FF	*First and Follow*
GTK	*Grammar Tool Kit*

Sumário

1 INTRODUÇÃO

Os cursos de graduação em Ciência da Computação frequentemente apresentam disciplinas que exigem uma capacidade elevada de abstração do aluno. Isso implica em uma alta demanda por ferramentas que auxiliem o aluno na absorção do conteúdo. Segundo Santos e Costa(2006), a falta de compreensão do raciocínio lógico pode ser o motivo pelo alto índice de reprovação nas disciplinas relacionadas a algoritmos e programação. Entre as disciplinas incluídas nesse grupo podemos acrescentar aquelas relacionadas ao ensino de compiladores.

O conteúdo teórico e prático estudado na disciplina de compiladores aborda as etapas que descrevem o processo de tradução de um programa escrito em uma linguagem de programação (denominada linguagem fonte) para outra linguagem (denominada linguagem alvo). Estas etapas são divididas didaticamente em duas fases: Análise (*front-end*) e Síntese (*back-end*). As etapas presentes na fase de análise são voltadas para a extração de informações do código fonte e dependem diretamente da estrutura gramatical deste código. As etapas presentes na fase de síntese são responsáveis pela produção do código equivalente escrito na linguagem alvo, com base nas informações adquiridas na fase anterior.

O trabalho de tradução é orientado pela descrição gramatical da linguagem. A gramática adotada no projeto de uma linguagem é essencial, pois é através dela que os sistemas de tradução identificam as entradas válidas e são capazes de extrair as informações necessárias. A gramática orienta todo o processo de tradução, por isso deve ser bem definida e livre de ambiguidade.

Observando a importância da construção gramatical no projeto de compiladores, este trabalho propõe uma solução em *software* para auxiliar a construção e correção de gramáticas livres de contexto disponibilizadas pelo usuário para serem utilizadas na construção de reconhecedores gramaticais baseados no método *top-down* preditivo (LL(1)).

1.1 Motivação e Justificativa

Ao longo das disciplinas oferecidas pela formação em Ciência da Computação no IFMG - Campus Formiga, notou-se uma dificuldade dos alunos no aprendizado dos conteúdos das disciplinas relacionadas com a teoria das linguagens de programação, particularmente as disciplinas Linguagens Formais e Autômatos, Teoria da Computação e Compiladores. Uma parte essencial no conteúdo dessas disciplinas é o estudo das gramáticas formais, particularmente aquelas livres de contexto. Segundo Aho et. al. (2008), a classe das gramáticas livres de contexto é rica o suficiente para reconhecer a maioria das construções

presentes nas linguagens de programação, mas escrever uma gramática adequada não é uma tarefa simples. Uma análise prévia constatou a existência de algumas ferramentas disponíveis para auxiliar a manipulação de gramáticas nessa área. Como exemplos das ferramentas encontradas, é possível citar:

- *Hacking Off: A Project Repository*: Um *website* que disponibiliza algumas ferramentas para a construção de compiladores, procurando automatizar algumas das monótonas técnicas necessárias para criação de um compilador. Descarta a necessidade de download de *software* pois funciona completamente online.

- **ANTLR** (*ANother Tool for Language Recognition*) - Um gerador de analisadores para descrição e processamento de linguagens, amplamente utilizado para construção de tradutores a partir de uma gramática.

Entretanto, estas ferramentas disponíveis possuem um objetivo diferente da proposta deste projeto, visto que quando identificam um erro, não auxiliam o usuário de uma forma didática (um detalhamento melhor destes *softwares* será realizado na seção 2.9 desta monografia). Além disso, apesar de possuírem o objetivo de tornar mais fácil a fase de projeto de compiladores, estas esperam que a gramática já esteja pronta, e não oferecem apoio na sua construção.

Visto a dificuldade de apresentar didaticamente o conteúdo, e a escassez de ferramentas que auxiliem o aprendizado, surge a necessidade de desenvolver instrumentos que melhorem a sua absorção. Isto posto, o presente trabalho contribui aumentando o arsenal de ferramentas disponíveis para apoiar o ensino de Compiladores.

1.2 Solução Proposta

Este trabalho tem como objetivo o desenvolvimento de um sistema que recebe como entrada gramáticas escritas pelo usuário numa linguagem formal definida neste trabalho, e disponibiliza uma série de recursos julgados importantes para a sua manipulação e ajuste. É previsto também que o *software* emita como saída um relatório apontando problemas e sugestões para a correção gramatical, proporcionando o auxílio em sua construção. Também é previsto que o *software* seja capaz de ajudar o aluno, não só na construção de gramáticas para a implementação de linguagens de programação, mas também no estudo para provas e exercícios da disciplina. Uma vez que é esperado que o sistema desenvolvido seja utilizado pelos alunos, é um requisito necessário que possua uma interface amigável e didática, para melhorar a absorção dos conteúdos teóricos e práticos relacionados a disciplina.

1.3 Objetivos

Esta seção tem como objetivo apresentar o objetivo geral que se deseja alcançar ao fim do trabalho bem como a construção dele será feita, através da conclusão de cada objetivo específico também apresentado.

1.3.1 Objetivo Geral

O objetivo deste trabalho é criar um sistema capaz de auxiliar os alunos da disciplina de compiladores na construção, teste e correção de gramáticas LL(1).

1.3.2 Objetivos específicos

- Descrever formalmente uma linguagem estilo Bacus-Naur-Form (BNF) para receber as linguagens que serão fornecidas para o sistema;
- Criar um compilador com os analisadores léxico, sintático e semântico que verifiquem a validade da entrada;
- Identificar os casos padrão que descrevem erros passíveis de apresentar uma solução correspondente.
- Implementar módulos que identificam erros e propõem correções para esses casos padrão. O sistema não promete correções para todo tipo de erro, visto que existem vários problemas indecidíveis desta área.
- Criar um módulo que disponibilize relatórios de correção de forma didática.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção serão apresentados toda fundamentação teórica para o desenvolvimento deste projeto, bem como os trabalhos relacionados ou aqueles que seguem a mesma linha de pensamento e desenvolvimento que auxiliaram e nortearam os procedimentos para a conclusão deste trabalho.

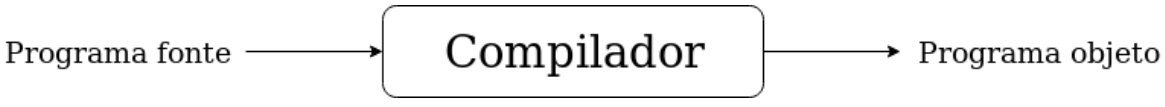
2.1 Linguagens de programação

Vieira (2006) define linguagens de programação como notações para se descrever computações para pessoas e para máquinas. Com a evolução da tecnologia assim como a sua importância no mundo atual, é possível notar que linguagens de programação são muito importantes, pois todo *software* em todos os computadores foi escrito em alguma linguagem de programação. Mas, antes que possa ser processado, um programa primeiro precisa ser traduzido para um formato que lhe permita ser executado por um computador. Esta seção mostra o que é responsável por esse processo de tradução, além de como ela é feita.

2.2 Compiladores

Uma definição sucinta de compilador, dada por Aho et. al. (2008), define um compilador como um programa que recebe como entrada um programa fonte escrito em alguma linguagem de programação, denominada como linguagem fonte. Em seguida o compilador o traduz para uma nova linguagem equivalente, denominada linguagem objeto. A Figura 1 mostra de forma superficial o papel de um compilador.

Figura 1 – Processo de tradução.



Fonte: Aho (2008)

Outro papel importante de um compilador é relatar quaisquer erros no programa fonte detectados durante esse processo de tradução. Tanto a detecção desses erros, quanto os processos utilizados para traduzir um programa estão presentes nas seções a seguir.

2.2.1 A estrutura de um compilador

A subseção 2.2 mostra o compilador como uma caixa-preta que mapeia um programa fonte para um programa objeto semanticamente equivalente. Abrindo esta caixa é possível ver que existem duas partes nesse mapeamento: análise e síntese.

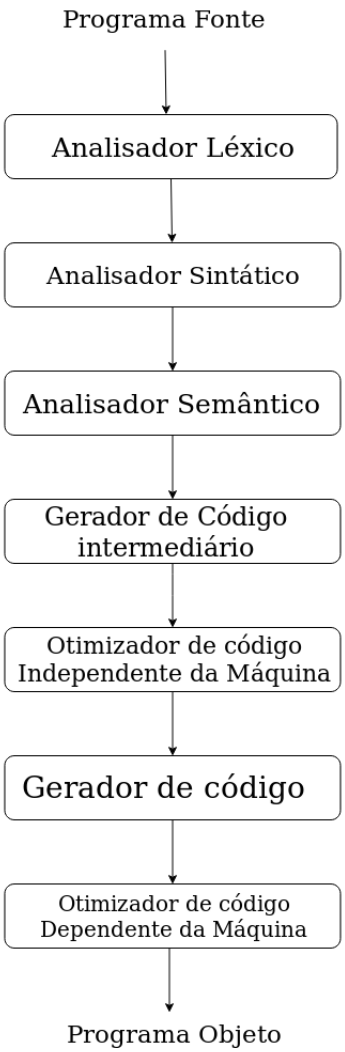
Aho et. al. (2008) define a fase de análise como um processo que subdivide o programa fonte em partes constituintes (análise léxica) e impõe uma estrutura gramatical sobre elas (análise sintática). Após isso, usa essa estrutura para criar uma representação intermediária do programa fonte. Se a fase de análise detectar que o programa fonte não segue a forma gramatical definida, ou está semanticamente incorreto (análise semântica), então o compilador deve oferecer mensagens esclarecedoras, de modo que o usuário possa tomar a ação corretiva.

Ainda segundo Aho et. al. (2008), a fase de síntese constrói o programa objeto desejado a partir da representação intermediária e das informações retiradas da fase de análise. Ao analisarmos todo o processo de compilação, é possível notar que ele é desenvolvido como uma sequência de fases. Cada uma transforma uma representação do programa fonte em outra, até que todo o processo de compilação esteja completo. A Figura 2 exibe a decomposição típica do processo de compilação em linguagens de programação.

A fase de análise normalmente é chamada de front-end do compilador. Ela inicia no analisador léxico e termina na fase denominada “gerador de código”; a parte de síntese é o back-end, representada pelas demais fases (Aho et. al. 2008). Este trabalho tem como foco a parte de análise do compilador, especificamente fase de análise sintática.

htb

Figura 2 – Fluxo típico de um compilador.



Fonte: Vieira (2008)

2.3 Análise Léxica

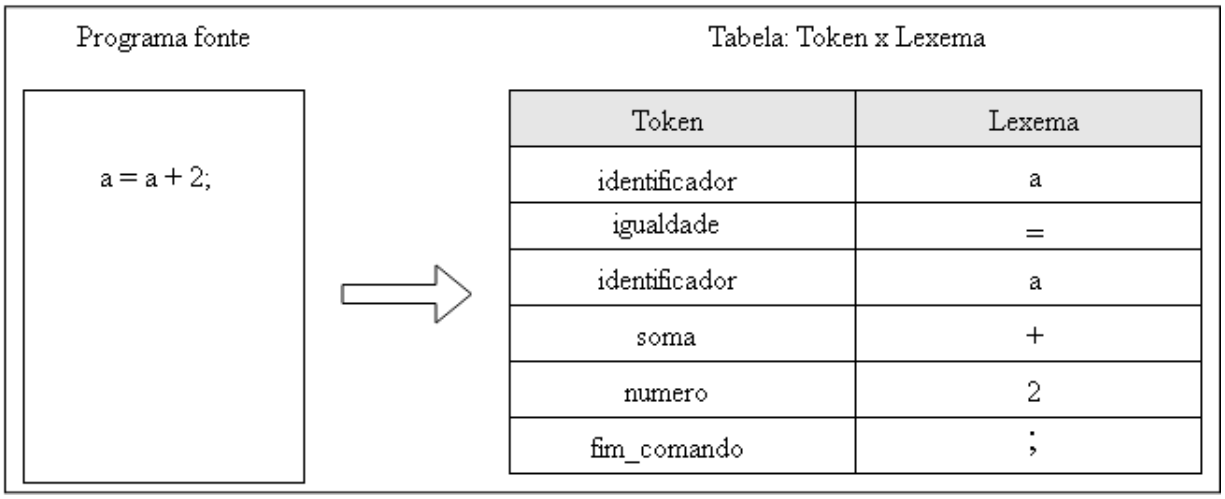
A análise léxica, como mostra a Figura 2, é a primeira fase do processo de compilação. Segundo Aho et. al. (2008), a principal tarefa de um analisador léxico é ler os caracteres da entrada do programa e agrupá-los em partes menores. As próximas subseções abordam como é feita a análise léxica, começando pela definição de *token*.

2.3.1 *Tokens* e lexemas

De acordo com Aho et. al. (2008), ao discutir a análise léxica, usamos dois termos relacionados, porém distintos, são eles *tokens* e lexemas. Um *token* é definido como um símbolo abstrato que representa um tipo de unidade léxica, por exemplo, um número, ou

um comando condicional (**if**). Já um lexema é definido como uma sequência de caracteres no programa fonte que se encaixa com o padrão de um *token*. A Figura 3 exemplifica a relação entre *token* e lexema, dado um programa fonte específico.

Figura 3 – Extração de lexemas.



Fonte: Autor

A Figura 3 mostra a extração dos lexemas de um programa fonte. Como exemplificado, *tokens* são atribuídos a cada lexema retirado do programa fonte, gerando a tabela "*Token* x Lexema", que será utilizada na fase de análise sintática do compilador.

É válido lembrar que geralmente os lexemas não são extraídos todos de uma vez, mas sim coletados um por um, sempre que o analisador sintático faz a requisição. Esta requisição é feita através da função *getToken()*.

Segundo Aho et. al. (2008), a função *getToken()* é a responsável por entregar ao analisador sintático, o próximo lexema extraído do código fonte. Ou seja, a cada chamada da função, o analisador léxico deve coletar o próxima cadeia de caracteres do código, atribuí-la ao seu determinado *token* e retornar um novo lexema a ser avaliado pelo analisador sintático.

2.4 Gramáticas Livre de Contexto

Para o entendimento da próxima fase de um compilador é necessário definir o conceito de Gramática Livres de Contexto (GLC). Segundo Vieira (2006), existem tipos diferentes de gramáticas, cada um desses tipos descreve um tipo de linguagem formal. Dentre as linguagens formais existentes é possível citar alguns exemplos como: linguagens regulares, linguagens sensíveis ao contexto e linguagens recursivas. Este trabalho possui

foco nas gramáticas voltadas para a construção de linguagens de programação usadas em compiladores, usualmente, as livres de contexto. Sendo assim, a subseção 2.4.1 apresenta, como introdução, o conceito de gramática formal. Em seguida, são mostrados as particularidades dessa determinada classe de gramáticas, as GLC's.

2.4.1 Gramáticas Formais

Vieira (2006) diz que gramáticas são um formalismo originalmente projetado para a definição de linguagens. Similarmente, uma gramática mostra como gerar as palavras de uma linguagem. Aho et. al. (2008) listam os quatro componentes que compõem as gramáticas:

- Um conjunto de não-terminais, denominados "variáveis sintáticas";
- Uma designação de um dos não-terminais como símbolo inicial da gramática.
- Um conjunto de símbolos terminais (*tokens*). Os terminais são símbolos elementares da linguagem, definidos pela gramática;
- Um conjunto de produções, também denominados regras.

Para demonstrar a geração de uma palavra de uma linguagem é necessário realizar a derivação desta palavra. Para isso, deve-se substituir as ocorrências de cada variável A que apareça na sentença por uma de suas produções. Vieira (2006) chama esse processo de **expansão de uma variável**. A Figura 4 apresenta uma gramática regular que possui S como variável inicial, apresenta uma gramática regular, bem como as produções presentes nela.

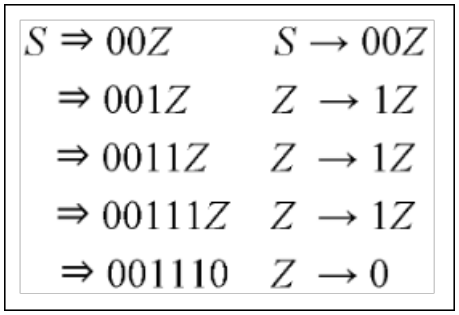
Figura 4 – Gramática Regular.

$$S \rightarrow 00Z$$
$$Z \rightarrow 1Z|\ 0Z\ |\ 0\ |\ 1$$

Fonte: Autor

A Figura 5 exemplifica a derivação da palavra w = 001110 presente na linguagem regular L = {w | w possui prefixo 00} e gerada pela gramática mostrada na Figura 4. Ou seja, neste caso, as palavras aceitas pela linguagem L serão apenas aquelas que apresentarem a cadeia "00"como prefixo.

Figura 5 – Exemplo Derivação.



Fonte: Autor

2.4.2 Características das Gramáticas Livres de Contexto

Quando se trata de gramáticas utilizadas em compiladores para a construção de linguagens de programação, é comum a utilização de gramáticas livres de contexto (GLC's).

Aho et. al. (2008) diz que por definição, as linguagens de programação possuem regras precisas para descrever a estrutura sintática de programas bem formados. A estrutura sintática das construções de uma linguagem é especificada pelas regras gramaticais de uma linguagem livre de contexto. O que diferencia as GLC's das de outras gramáticas é o fato de que são permitidas apenas regras em que o lado esquerdo de cada símbolo de produção apresenta apenas uma variável. Ou seja, $S \Rightarrow Au$ é uma produção válida em GLC's, e $BA \Rightarrow au$ não é uma notação pertencente a este tipo de gramática.

Vieira (2006) define formalmente uma gramática livre de contexto (GLC) como uma gramática (V, Σ, R, P), em que cada regra tem a forma $X \Rightarrow w$, em que $X \in V$ e $w \in (V \cup \Sigma)*$. O conjunto V é composto pelas variáveis sintáticas da gramática. O conjunto Σ é composto pelos terminais presentes na gramática. Já o conjunto R é formado pelas regras de produção. Por fim, P define qual das variáveis é dita inicial.

Um exemplo retirado do livro 'Compiladores: Princípios, técnicas e ferramentas' de Aho et. al. (2008), apresenta a gramática livre de contexto $(E, T, F, t, +, *, (,), R, E)$, utilizada em expressões aritméticas, em que R consta das regras mostradas na Figura 6.

Figura 6 – Expressões aritméticas e regras de produção da gramática.

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

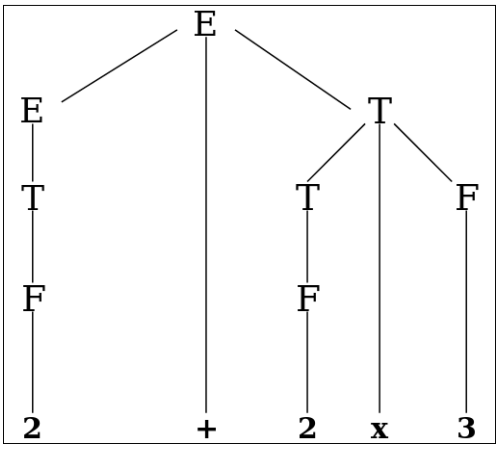
$$F \rightarrow (E) \mid t$$

Fonte: Aho et. al. (2008)

A Figura 7, mostra como a gramática da Figura 6 gera a expressão aritmética 2+2∗3, considerando t como terminais numéricos pertencentes aos inteiros. Esta demonstração em forma de árvore é definida como árvore de derivação, em que Aho define da seguinte maneira:

> "Uma árvore de derivação é uma representação gráfica de uma derivação que filtra a ordem na qual as produções são aplicadas para substituir não-terminais. Cada nó interior de uma árvore de derivação representa a aplicação de uma produção. O nó interior é rotulado como o não-terminal A do lado esquerdo da produção; os filhos deste nó são rotulados, da esquerda para a direita, pelos símbolos do corpo da produção pelo qual esse A foi substituído durante a derivação."[Aho et. al. 2008]

Figura 7 – Árvore de Expansão para uma GLC.



Fonte: Aho et. al. (2008)

Observando a Figura 7, é visto que as GLC's possuem a capacidade de realizar a precedência de operadores. Nota-se que o operador de multiplicação foi programado para ter maior precedência que o operador de soma. Esta é uma característica que faz com

que as GLC's sejam bastante aplicadas em projetos de compiladores para linguagens de programação.

2.4.3 Ambiguidade em Gramáticas Livres de Contexto

Vieira (2006) define que uma GLC é denominada ambígua quando existe mais de uma árvore de derivação para alguma sentença que ela gera. É importante observar que a gramática é dita ambígua, afinal, podem haver outras GLC's equivalentes a uma GLC ambígua, que não apresente esta característica.

Um exemplo dado por Vieira (2006) no livro *"Introdução aos Fundamentos da Computação"*, apresenta a gramática $G = (E, t, +, *, (,), R, E)$, escrita para realizar expressões aritméticas, em que o conjunto R consta das regras apresentadas na Figura 8.

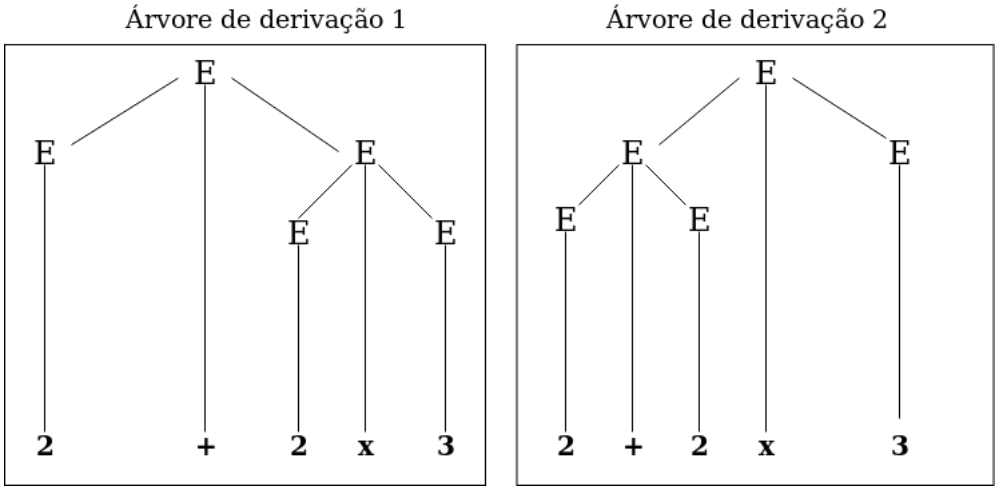
Figura 8 – Exemplo de gramática ambígua.

$$E \rightarrow E + E \mid E * E \mid (E) \mid t$$

Fonte: Vieira (2006)

A gramática mostrada pela Figura 8 é ambígua. Substituindo o terminal t como qualquer valor numérico inteiro, existem duas árvores de derivação (AD's) para a palavra $2 + 3 * 3$, como mostra a Figura 9.

Figura 9 – Prova de ambiguidade.



Fonte: Aho et. al. (2008)

Como consequência, a possibilidade de gerar duas AD's para a mesma sentença pode gerar problemas. Nesse caso, é possível observar na Figura 9 um problema de precedência de operadores, já que a gramática não está bem definida quanto a qual operador ($+$ ou x) possui maior prioridade ao realizar as operações.

2.5 Análise Sintática

Após abordar os conceitos que envolvem a primeira fase de um compilador (subseção 2.3) e discorrer sobre as GLC's, esta seção apresenta a segunda fase do compilador, denominada análise sintática.

Segundo Delamaro (2004), no modelo atual de compilador, o analisador sintático tem como objetivo receber uma sequência de *tokens* disponibilizados pelo analisador léxico e aplicá-la em uma estrutura gramatical. Além disso, o analisador sintático deve ser projetado para emitir mensagens de erro para quaisquer erros de sintaxe encontrados no programa fonte, ou seja, receber umas sequência de *tokens* que não se encaixa na gramática determinada pelo analisador.

Segundo Aho et. al. (2008), existem três estratégias gerais de análise sintática para o processamento de gramáticas: universal, descendente e ascendente. Os modelos de análise baseados na estratégia universal podem analisar qualquer gramática. No entanto, esses modelos são muito ineficientes para serem usados com compiladores. Sendo assim, os métodos geralmente escolhidos para a construção de compiladores são os baseados nas estratégias descendentes ou ascendentes. O presente trabalho propõe uma ferramenta que auxilie na construção de GLC's da classe LL(1) , utilizadas na estratégia descendente, que será abordada na seção 2.7 deste documento.

2.6 Conjuntos First e Follow

Aho et. al. (2008) diz que a construção de analisadores descendentes é auxiliada por duas funções, *First* e *Follow* (FF), associadas a uma gramática. Durante a análise descendente as funções FF nos permitem escolher qual produção aplicar, com base no próximo lexema disponibilizado pelo analisador léxico. A seguir serão dados os detalhes sobre ambas as funções.

2.6.1 Função *First*

A função *First* tem como objetivo construir um conjunto composto pelos primeiros lexemas (mais a esquerda) que uma determinada variável pode consumir. A Tabela 1 mostra a aplicação da função *First* para todas as variáveis presentes na gramática expressada pela Figura 10.

Figura 10 – Gramática para cálculo de First set.

$$S \rightarrow A\ c \mid b\ B$$
$$A \rightarrow a\ A \mid \lambda$$
$$B \rightarrow b\ B \mid b$$

Fonte: Autor

Tabela 1 – Cálculo do conjunto *First*.

Variável	First
S	a b c
A	a λ
B	b

Fonte: Autor

De acordo com a Figura 10 é possível afirmar que:

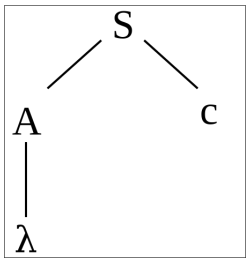
- A variável B possui apenas o terminal b em seu conjunto *First*, afinal em todas as suas regras, b aparece como primeiro lexema.

- A variável A por sua vez, apresenta os símbolos a e λ em seu conjunto *First*. A ocorrência de a em *First*(A) se justifica, pois a regra $A \Rightarrow aA$ apresenta o lexema a como primeiro símbolo a ser consumido. De forma semelhante, λ está contido em seu conjunto, pois a regra $A \Rightarrow \lambda$ nos diz que a variável A pode derivar uma cadeia vazia.

- A variável S possui b em seu conjunto *First*, justificado pela regra $S \Rightarrow bB$. A ocorrência de a em *First*(S) se justifica pelo fato de que a regra $S \Rightarrow Ac$ provoca a derivação de A inicialmente, podendo a aparecer como primeiro símbolo da variável S. Por fim, c também está presente em *First*(S). Isso se justifica através da regra $S \Rightarrow Ac$. Nessa regra, A é a primeira derivação realizada, como a variável A pode derivar uma cadeia vazia, nesse caso, c pode ser o primeiro caractere consumido pela variável S.

A Figura 11 a seguir apresenta uma AD que exemplifica melhor a ocorrência de c em *First*(S).

Figura 11 – 'c' em First de S.



Fonte: Autor

Como mostrado pela Figura 11, por possuir a regra $A \Rightarrow \lambda$ a variável A pode não gerar terminais, nesse caso, c é o primeiro terminal a ser consumido por S, portanto c está contido em $First$(S).

2.6.2 Função *Follow*

A função *Follow* tem como objetivo construir o conjunto de terminais que podem aparecer após a derivação de uma determinada variável. A Tabela 2 mostra o cálculo dos conjuntos *Follow* das variáveis da gramática apresentada na Figura 10.

Tabela 2 – Cálculo do conjunto *Follow*.

Variável	*Follow*
S	$
A	c
B	$

Fonte: Autor

Como apresentado na Tabela 2, a variável A possui o lexema c em seu conjunto *Follow*, isso se justifica devido ao fato de que a ocorrência de A na regra $S \Rightarrow Ac$, possui o terminal c como primeiro lexema a ser consumido após sua derivação. Analogamente, a mesma estratégia é aplicada a todos os casos, se atentando ao fato de que, por ser a variável inicial, S possui $ em seu conjunto Follow. Isso ocorre devido ao fato de que $ representa o fim do arquivo de entrada, informando que a cadeia de caracteres chegou ao fim.

2.7 Análise LL(1)

De acordo com Aho et. al. (2008), o método de análise descendente constrói a árvore de derivação para a cadeia de entrada de cima para baixo, coletando os *tokens* a partir do primeiro caractere apresentado pelo programa fonte. Este método possui várias abordagens para tratar os *tokens* recebidos pelo analisador léxico, definidas a partir do tipo de GLC construída para o projeto da linguagem. Porém, a abordagem mais comum utilizada em linguagens de programação é a que utiliza GLC's do tipo LL(1).

Aho et. al. (2008) dizem que a classe de GLC's do tipo LL(1) são aplicadas a analisadores sintáticos preditivos, ou seja, aqueles que não precisam de realizar retrocessos na sequência de *tokens* recebida pelo analisador léxico. O primeiro 'L' em LL(1) significa que a cadeia de *tokens* é avaliada da esquerda para a direita(L = *Left-to-Right*); o segundo 'L' representa uma derivação mais à esquerda(L = *Leftmost*), ou seja a AD busca sempre expandir o nó disponível mais a esquerda; e o '1' define que sempre será analisado um *token* por vez.

Segundo Aho et. al. (2008), uma GLC é LL(1) se somente se, sempre que $A \Rightarrow a|B$ forem duas produções distintas de G, as seguintes condições forem verdadeiras:

1. Para um terminal a, tanto a quanto B não derivam cadeias cadeias começando com a;
2. No máximo um dos dois, a ou B, pode derivar a cadeia vazia;
3. A gramática não pode ser ambígua;
4. A gramática não deve apresentar recursão à esquerda em nenhuma de suas variáveis.

As subseções a seguir dão embasamento teórico para compreender essas condições apresentadas por Aho et. al. (2008).

2.7.1 Recursão à esquerda

Aho et. al. (2008) define formalmente que uma variável apresenta uma recursão à esquerda se existe uma derivação $A \Rightarrow Aa$ em alguma de suas regras. Em outras palavras é possível dizer que se uma variável A possui uma regra que apresente A como primeira derivação, esta se mostra recursiva a esquerda. A Figura 12 apresenta uma GLC não-LL(1), pois as variáveis A e B possuem regras recursivas a esquerda.

Aho et. al. (2008) dizem que métodos descendentes não podem tratar gramáticas com recursão à esquerda, de modo que uma manipulação gramatical é necessária para eliminar a recursão.

Figura 12 – Exemplo recursividade à esquerda.

$$A \rightarrow A\ c \mid B$$
$$B \rightarrow Bd \mid d$$

Fonte: Autor

2.7.2 Indeterminação entre regras de uma gramática

Hopcroft (2001) diz que durante o processo de construção da gramática, o cálculo dos conjuntos FF mostra a sua importância. Uma das funcionalidade que esse cálculo apresenta é a descoberta de indeterminações na gramática (seção 2.7.2). A Tabela 3 que apresenta o cálculo dos conjuntos FF da gramática mostrada pela Figura 13, aborda um problema em gramáticas LL(1) . Quando uma variável pode derivar uma cadeia vazia, é indispensável que esta não apresente uma intercecção entre seus conjuntos *First* e *Follow*.

Figura 13 – Gramática com inresecção entre *First* e *Follow*.

$$S \rightarrow A\ a \mid c$$
$$A \rightarrow a\ A \mid \lambda$$

Fonte: Autor

Tabela 3 – Cálculo *First* e *Follow*.

Variável	*First*	*Follow*
S	a c	$
A	a λ	a

Fonte: Autor

A Figura 3 apresenta uma intersecção entre os conjuntos FF da variável A. Nesse caso, não fica claro para o analisador sintático se a variável A deve consumir o terminal a, ou derivar uma cadeia vazia, deixando que os próximos passos da derivação consumam este terminal. É dito portanto que existe uma indeterminação na gramática. Aho et. al. (2008) alertam no livro *"Compiladores, técnicas e ferramentas"* que nem sempre é possível corrigir este problema nos casos dos comandos condicionais (**if** e **else**). Neste caso, e apenas

neste caso, o problema é corrigido na codificação do analisador sintático, não realizando alterações na gramática.

Outro problema que gera indeterminação é abordado por Aho et. al. (2008). Em analisadores descendentes (ou preditivos) a indeterminação entre regras de uma mesma variável não pode ocorrer. Em gramáticas LL(1) por exemplo, em que o analisador sintático avalia um símbolo por vez, a escolha entre duas ou mais alternativas deve ser clara, não apresentando regras com a mesma sentença inicial. A Figura 14 mostra um exemplo de indeterminação, neste caso a gramática não é LL(1), sendo imprópria para a implementação de um analisador preditivo.

Figura 14 – Indeterminação entre regras.

$$S \rightarrow a\ S \mid a\ c$$

Fonte: Autor

2.8 Simplificação de Gramáticas Livres de Contexto

Esta seção aborda as diferentes formas de simplificar uma GLC. Hopcroft (2001) aborda dois principais motivos para simplificar uma gramática livre de contexto:

- A aplicação da simplificação pode melhorar a legibilidade, facilitando a leitura da gramática por parte do seu construtor;
- Após ter a gramática construída, o próximo passo é a construção do analisador sintático (*parser*). Neste momento, em que a legibilidade já não é importante, a simplificação gramatical é útil ao transformar uma gramática para que a mesma esteja em um formato mais adequado para o processamento realizado no *parser*.

Sendo assim, as subseções a seguir abordam as simplificações julgadas importantes pela literatura utilizada como referência para este trabalho.

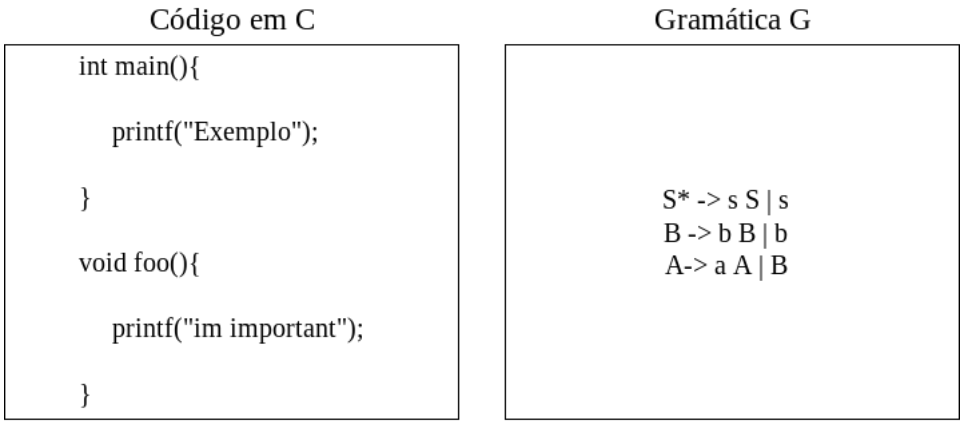
2.8.1 Variáveis Inúteis

Segundo Vieira (2006) a detecção de variáveis que nunca participam de derivações de palavras da linguagem gerada por uma GLC, as chamadas variáveis inúteis, é importante por vários motivos. Por exemplo, em gramáticas grandes, como as de linguagens de programação, as regras de uma variável A podem ter sido definidas, mas A não foi utilizada em nenhuma das regras de outra variáveis. Caso A seja a variável inicial da

gramática, então ela não é reconhecida como inútil, afinal ela é chamada no início da derivação. Do contrário, A jamais será invocada, tornando assim inútil.

É possível realizar um paralelo entre variáveis inúteis e funções (procedimentos) desnecessários em um programa. A Figura 15 demonstra esse paralelo.

Figura 15 – Paralelo entre funções e variáveis gramaticais.



Fonte: Autor

No lado esquerdo(Código em C) da Figura 15 é possível ver um código fonte escrito na linguagem de programação C. Nele é possível observar que o método *foo* não terá qualquer impacto quando o código for executado, afinal este não foi utilizado no método main do código fonte. É importante perceber que o método main() também não foi chamado em nenhuma função, mas este é tido como inicial ao executar qualquer código escrito em C.

No lado direito (Gramática livre de contexto) da Figura 15 é apresentada uma GLC em que a variável A é inútil, pois nunca participará da derivação a partir da variável inicial S. Nota-se que S se comporta como o método main presente no código da Figura 15, pois não existe a necessidade de ser chamada em regras de outras variáveis sendo que S é a variável inicial da gramática.

Logo, assim como o método *foo* pode ser descartado do código fonte, a variável A também pode ser excluída da gramática. Por fim é válido lembrar que a exclusão de uma variável de uma gramática pode desencadear uma exclusão em cascata, afinal, como podemos ver na gramática da Figura 15, ao excluirmos a variável A, a variável B também entraria no grupo das variáveis inúteis.

2.8.2 Variáveis Inférteis

Existem outros tipos de variáveis que são passíveis de exclusão em uma gramática. A Figura 16 apresenta uma gramática que possui a variável C como infértil.

Esse tipo de variável é passível de exclusão pois nunca irá gerar terminais, pois apresenta recursividade em todas as suas regras. Além disso, todas as regras que possuem C como possível derivação também devem ser eliminadas. A Figura 17 apresenta a gramática resultante após a exclusão da variável C.

Figura 16 – Exemplo de gramática com variáveis inférteis.

$$S \rightarrow a\ S\ b \mid c\ C \mid ab$$
$$C \rightarrow cC \mid dC$$

Fonte: Autor

Figura 17 – Variável C eliminada.

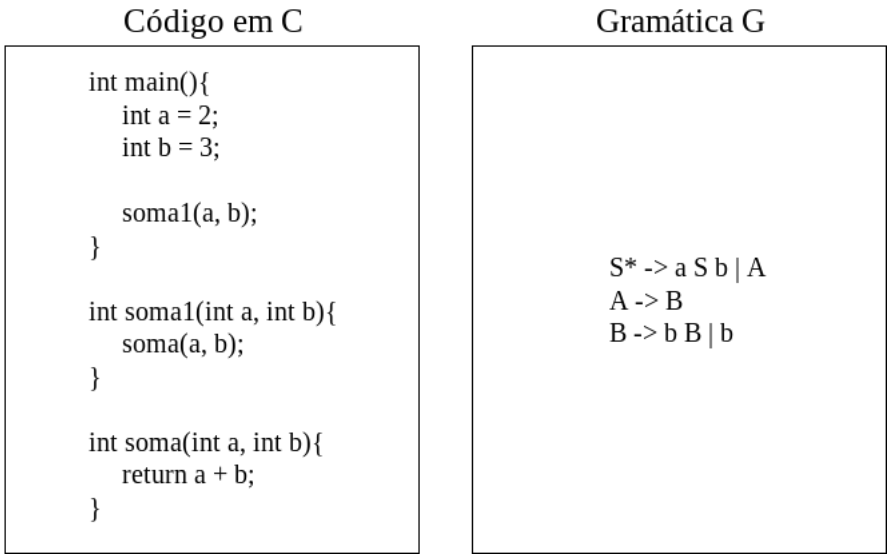
$$S \rightarrow a\ S\ b \mid ab$$

Fonte: Autor

2.8.3 Produções Unitárias

Outra simplificação que pode ser feita em GLC's é a remoção de produções unitárias. Produções unitárias são aquelas que possuem o formato $A \Rightarrow B$. Neste caso é possível dizer que o projetista da gramática optou por essa abordagem com o intuito de obter legibilidade, mas para o *parser* as produções unitárias são um passo adicional a ser realizado durante o processamento. A Figura 18 mostra um paralelo entre uma gramática com produções unitárias e um código de programação escrito em C.

Figura 18 – Variável C eliminada.



Fonte: Autor

No código apresentado na Figura 18 é possível observar que o programa realizará a soma de a e b, mas passar pela função $soma1$ é desnecessário, afinal ela não realiza nenhuma tarefa adicional, apenas chama a função soma. Assim como no código, é possível notar que na gramática G presente na Figura 18 uma mesma abordagem. A variável A possui uma regra que apenas chama a derivação da variável B. Neste caso, a gramática poderia ser projetada para chamar B diretamente sempre que necessário, poupando o *parser* de realizar uma chamada adicional.

2.9 Trabalhos Relacionados

Esta seção aborda os trabalhos relacionados ao projeto desenvolvido. É importante comentar que, mediante as pesquisas feitas, não foram encontrados trabalhos que possuem foco na construção gramatical de um compilador. Os trabalhos encontrados possuem algumas funções em comum com este projeto, mas se desviam da fase de construção gramatical, apresentando funções que fogem do escopo deste trabalho de conclusão de curso.

2.9.1 *Hacking Off*

É um site auto denominado como uma ferramenta de construção de compiladores. O *Hacking Off* oferece algumas das operações úteis na criação de um compilador, procurando

automatizar algumas das monótonas técnicas necessárias. Descarta a necessidade de *download* de *software* pois funciona completamente *online*.

Dentre as funções prometidas pelo *site*, estão presentes:

- Conversão de autômatos finitos não determinísticos, para autômatos finitos determinísticos;
- Transformação de uma expressão regular, para autômato finito;
- Gerador de *Scanners* para a captura de *tokens*;
- Cálculo dos conjuntos *First* e *Follow* em gramáticas LL(1).

O cálculo dos conjuntos *First* e *Follow* é o único método em comum entre o *Hacking Off* e este trabalho de conclusão de curso.

2.9.2 ANTLR (*ANother Tool for Language Recognition*)

O ANTLR é um poderoso gerador de analisador que você pode usar para ler, processar, executar ou traduzir arquivos estruturados de texto ou binários. É utilizado tanto na academia quanto na indústria, para construir todo tipo de linguagem, ferramentas e *frameworks*.

O ANTLR é capaz de receber comandos e expressões regulares para a construção de analisadores léxicos. Também é capaz de criar analisadores sintáticos partir de uma gramática.

Apesar de muito útil na criação de linguagens de programação, o ANTLR não ajuda no processo de criação gramatical. Ele considera que esse processo já foi realizado, disponibilizando ferramentas para a construção da linguagem, mas considerando que toda a parte de planejamento já foi realizada.

3 MATERIAIS E MÉTODOS

Esta seção apresenta tudo o que foi utilizado no desenvolvimento do trabalho, assim como detalhes de configurações no caso de possível replicação, além de uma subseção (3.2) para descrição da metodologia composta por todos os detalhes relevantes para o entendimento do norte seguido pelo projeto.

3.1 Materiais

Esta subseção diz respeito a prospecção de todos os *hardwares* e *softwares* utilizados na confecção deste projeto, bem como toda a configuração original dos mesmos, e toda configuração necessária feita neles, para o funcionamento do *software* proposto.

3.1.1 Configuração dos *hardwares* utilizados

Os materiais utilizados estão descritos e detalhados nas Tabela 4 a seguir, nela está contido o ambiente de desenvolvimento deste trabalho.

Tabela 4 – Máquina de implementação e testes do *software*.

Item	**Descrição**
Hardware	• Notebook Dell Inspiron 14 7000, Modelo 7460 • Processador Intel Core i5-4200U CPU@ 1.60Hz 2.30Hz, 3,00 MB de Memória Cache. • Memória Ram: 8,00 GB. • SSD: 256 GB.
Sistema Operacional	Linux Xubuntu 16.04 64 bits.
Compilador/Interpretador	Python 2.7.
Ambiente	Sublime Text.

Fonte: Autor

3.1.2 Linguagem de programação

Este trabalho foi desenvolvido inteiramente em sistema Linux Ubuntu (com interface *xFce*), e a codificação na linguagem de programação Python (versão 2.7). A justificativa vem do interesse em linguagem de fácil e rápida implementação, já que esta possui

uma codificação simplificada além de disponibilizar diversos recursos já implementados e disponibilizados ao programador. Há controvérsias quanto a velocidade de processamento dos códigos em Python, mas não convém para este trabalho, já que a complexidade do mesmo está voltada para a implementação, e não ao processamento do código.

3.2 Metodologia

Esta seção tem como objetivo relatar a metodologia utilizada para a realização deste trabalho. A próximas subseções apresentam as etapas necessárias para alcançar os objetivos propostos.

3.2.1 Preparação

Primeiramente foi necessário estudar os problemas recorrentes em gramáticas livre de contexto LL(1). Foi realizado um estudo acerca das obras com o conteúdo direcionado a linguagens formais e projeto de compiladores. Em seguida foram selecionados os algoritmos relevantes presentes nessas obras. Também foi planejada a construção de outros algoritmos julgados importantes, com base nos problemas recorrentes da disciplina de compiladores. Esses algoritmos serão detalhados na subseção 3.2.2 (Algoritmos) a seguir.

Em segundo lugar, foram coletados *cases* para testes, presentes nas obras referenciadas neste documento e listas de exercícios da disciplina de compiladores disponibilizadas nos acervos de faculdades federais. Os *cases* serão exibidos na subseção 4.6, e os resultados encontrados serão exibidos na seção 5.

Em seguida foi realizado o estudo dos analisadores léxico e sintático, necessários para a implementação da linguagem no formato BNF que irá receber como entrada as gramáticas fornecidas pelo usuário. Também foi revisado os conceitos que envolvem a criação de GLC's, necessários para construir a gramática dessa linguagem. A criação desta será detalhada na seção 4.2 deste documento.

Por fim, foram estudados as possíveis formas da saída, que será disponibilizada ao usuário. Para isto, foram estudadas possíveis interfaces gráficas disponíveis para a linguagem Python, além dos *softwares* presentes no Linux, que utilizam o terminal como interface para o usuário.

3.2.2 Algoritmos

Apresentados os problemas recorrentes em GLC's nas seções 2.7 e 2.8, a pesquisa realizada acerca da literatura permitiu dar os passos iniciais para a implementação dos algoritmos necessários. As subseções a seguir mostram os algoritmos escolhidos para serem implementados no *software* e onde eles foram encontrados.

3.2.2.1 Cálculo dos conjuntos *First* e *Follow*

A subseção 2.6 apresenta a importância do cálculo dos conjuntos First e Follow para o projeto de um compilador. Além do que foi apresentado, é importante também registrar a sua importância no aprendizado em sala de aula, afinal, é um processo cobrado em provas e exercícios da disciplina.

Disponível no livro "Como construir um Compilador - Utilizando Ferramentas Java" de Delamaro (2004), os algoritmo escolhidos para a implementação do cálculo dos conjuntos First e Follow são mostrados pelas figuras 19 e 20 respectivamente.

Figura 19 – Algoritmo cálculo conjunto *First*

- para o terminal α, $FIRST(\alpha) = \{\alpha\}$;
- para o não-terminal B tal que $B \rightarrow A_1 A_2 \ldots A_n$, onde A_i são símbolos terminais ou não-terminais, faz-se:
  - inicialmente $FIRST(B) = \{\}$
  - repete-se $FIRST(B) = FIRST(B) \cup FIRST(A_i)$, para $i = 1, 2, \ldots$ até que se encontre algum i tal que A_i não deriva λ;
- para uma string $a = A_1 A_2 \ldots A_n$, onde A_i são símbolos terminais ou não terminais, faz-se:
  - repete-se $FIRST(\alpha) = FIRST(\alpha) \cup FIRST(A_i)$, para $i = 1, 2, \ldots$ até que se encontre algum i tal que A_i não deriva λ;

Fonte: Delamaro (2004)

Figura 20 – Algoritmo cálculo conjunto *Follow*

- adiciona-se o indicador de fim de cadeia \$ ao conjunto $FOLLOW(S)$, onde S é o símbolo inicial da GLC;
- dada a produção $B \rightarrow \alpha A \beta$, faz-se:
  - $FOLLOW(A) = FOLLOW(A) \cup FIRST(\beta)$
  - se $\beta \Rightarrow^* \lambda$, então faz-se $FOLLOW(A) = FOLLOW(A) \cup FOLLOW(B)$.

Fonte: Delamaro (2004)

3.2.2.2 Fatoração à esquerda

O algoritmo de fatoração à esquerda tem como objetivo remover as indeterminações entre regras. No fim das iterações espera-se que a gramática não possua indeterminações, se aproximando do formato LL(1).

O presente na Figura 21 foi escolhido para a implementação desta função, e está disponível no livro "Compiladores: Princípios, técnicas e ferramentas" de Aho et. al. (2008). Nota-se que este não se trata de um pseudocódigo convencional, mas de uma descrição formal de como remover as regras que causam a indeterminação.

Figura 21 – Algoritmo fatoração à esquerda

ENTRADA: Gramática G.

SAÍDA: Uma gramática equivalente a G, fatorada à esquerda.

MÉTODO: Para cada não-terminal A, encontre o prefixo α mais longo, comum a duas de suas alternativas. Se $\alpha \neq \varepsilon$ - ou seja, existe um prefixo comum não-trivial-, substitua todas as produções- A, $A \rightarrow \alpha\beta_1|\alpha\beta_2|...|\alpha\beta_n|\gamma$, onde γ representa todas as alternativas que não começam com um α, por:

$$A \rightarrow \alpha\, A' \mid \gamma$$
$$A' \rightarrow \beta_1|\beta_2|...|\beta_n$$

A' representa um novo não-terminal. Aplique repetidamente essa transformação até que não haja duas alternativas para um não-terminal com um prefixo comum.

Fonte: Aho et. al. (2008)

3.2.2.3 Eliminação de Recursão à Esquerda

Outro algoritmo julgado importante para a construção de GLC's é o algoritmo de eliminação de recursão à esquerda, que é um problema que impede que uma GLC seja classificada como LL(1). O pseudocódigo presente na Figura 22 foi denotado por Aho et. al (2008).

3.2.2.4 Eliminação de Produções Unitárias

Abordado na seção 2.8.3 deste documento, outro algoritmo implementado é o de eliminação de produções (ou regras) unitárias. O seguinte pseudocódigo mostrado pela Figura 23 foi extraído do livro "Introdução aos fundamentos da computação" de Vieira (2006).

Figura 22 – Algoritmo de remoção de recursão à esquerda

ENTRADA: Gramática G.

SAÍDA: Uma gramática equivalente a G, sem recursão à esquerda.

PSEUDOCÓDIGO:

ordene os não-terminais em uma ordem crescente qualquer $A_1, A_2, \ldots, A_n$.
para cada i de 1 até n {
  para cada j de 1 até i-1{
    substitua cada produção da forma $A_i \rightarrow A_j\gamma$ pelas
      produções $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma|\ldots|\delta_k\gamma$, onde
      $A_j \rightarrow \delta_1 \mid \delta_2|\ldots|\delta_k$ são produções-A_j
  }
  elimine as recursões esquerdas imediatas nas produções-A_i
}

Fonte: Aho et. al. (2008)

Figura 23 – Algoritmo de remoção de produções unitárias

ENTRADA: uma GLC $G = (V, \Sigma, R, P)$;

SAÍDA: uma GLC G' equivalente a G, sem produções diretas.

PSEUDOCÓDIGO:

$R' \leftarrow \varnothing$
para cada variável $X \in V$ {
  $R' \leftarrow R' \cup \{X \rightarrow \omega \mid$ existe $Y \in enc(X)$ tal que $Y \rightarrow \omega \in R$ e $\omega \notin V\}$
}
retorne $G' = (V, \Sigma, R', P)$.

Fonte: Vieira (2006)

3.2.2.5 Detecção de Variáveis Férteis

O algoritmo mostrado pela Figura 24 tem como objetivo detectar as variáveis férteis de uma gramática. Tendo este em mãos, foi possível implementar o algoritmo de remoção

de variáveis inférteis. Este algoritmo, utilizado para a implementação do *software* proposto por este trabalho, foi denotado por Vieira (2006).

Figura 24 – Algoritmo de detecção de variáveis férteis

Entrada: uma GLC $G = (V, \Sigma, R, P)$.

Saída: conjunto das variáveis de G que produzem sentenças.

PSEUDOCÓDIGO:

$I \leftarrow \varnothing$;

repita

$N \leftarrow \{X \notin I \mid X \rightarrow z \in R \text{ e } \in (I \cup \Sigma)^*\}$

$I \leftarrow I \cup N$

até $N = \varnothing$;

retorne I.

Fonte: Vieira (2006)

3.2.2.6 Detecção de variáveis alcançáveis

Após abordar os conceitos sobre variáveis inúteis na seção 2.8.1, o algoritmo presente na Figura 25, disponibilizado por Vieira (2006), tem como objetivo encontrar as alcançáveis a partir da variável inicial da gramática. A partir deste pseudocódigo, foi possível implementar o algoritmo de remoção de variáveis inalcançáveis.

Figura 25 – Algoritmo de detecção de variáveis alcançáveis

Entrada: uma GLC $G = (V, \Sigma, R, P)$.

Saída: conjunto das variáveis de G que são alcançáveis a partir da variável inicial de G.

PSEUDOCÓDIGO:

$I \leftarrow \varnothing$; $N \leftarrow \{P\}$

repita

$I \leftarrow I \cup N$

$N \leftarrow \{Y \notin I \mid X \rightarrow \mu Y \nu$ para algum $X \in N$ e $\mu, \nu \in (V \cup \Sigma)^*\}$

até $N = \varnothing$;

retorne I.

Fonte: Vieira (2006)

3.2.2.7 Outros algoritmos

Além dos algoritmos apresentados, outras três funções foram criadas. Como São elas:

- Remoção de regras duplicadas: Detecta regras repetidas em uma gramática. Por exemplo: $A \Rightarrow aA|aA$;
- Remoção de variáveis duplicadas: Detecta variáveis idênticas em uma gramática. por exemplo: $A \Rightarrow cC|d$; $B \Rightarrow cC|d$;
- Detecção de variáveis não declaradas: Detecta variáveis que foram utilizadas, mas não foram declaradas.

3.2.3 Saída de dados

Por se tratar de um *software* que busca o auxiliar o aluno, foi decidido mostrar ao usuário um passo-a-passo das alterações que a gramática vai sofrendo durante os algoritmos. Inicialmente foi considerado implementar uma interface gráfica, exibindo a saídas dos dados nessa interface. Mas ao ver que seria necessário mais tempo para o desenvolvimento, optou-se por manter a interface com o usuário através da linha de comando do GNU/Linux e registrar a saída dos dados através de uma página HTML gerada pelo *software*. A Figura 26 mostra um exemplo de uma saída gerada.

Figura 26 – Exemplo de saída gerada.

GTK - Grammar Tool Kit 1.0

Remove left Recursion

Steps	
<S> -> <S> a | <C> <C> -> <C> c | c	(1)
<S> -> <C> | <C> <S'> <C> -> <C> c | c <S'> -> a | a <S'>	(2)
<S> -> <C> | <C> <S'> <C> -> <C> c | c <S'> -> a | a <S'>	(3)
<S> -> <C> | <C> <S'> <C> -> c | c <C'> <S'> -> a | a <S'> <C'> -> c | c <C'>	(4)
<S> -> <C> | <C> <S'> <C> -> c | c <C'> <S'> -> a | a <S'> <C'> -> c | c <C'>	(5)

Fonte: Autor

Como exemplo, foi executado o algoritmo de remoção de recursividade à esquerda.

1. O primeiro quadro apresenta a gramática de entrada. A cor vermelha em $< S >$, mostra que a mesma possui recursividade à esquerda;

2. O próximo quadro mostra em azul as alterações realizadas para que $< S >$ não possua mais recursividade à esquerda;

3. Semelhante ao quadro (1), o quadro (2) assinala que a variável $< C >$ possui possui recursividade à esquerda;

4. Semelhante ao quadro (2), o quadro (4) apresenta as alterações que removem a recursividade de $< C >$;

5. Por fim, a gramática colorida em verde mostra que está livre de recursão a esquerda.

4 DESENVOLVIMENTO

4.1 O *software* GTK

Como dito anteriormente este trabalho tem como objetivo a criação de um *software* com o objetivo de servir de ferramenta na criação de linguagens de programação LL(1) e prover auxílio no aprendizado da disciplina de compiladores. Por ter este propósito, o mesmo foi nomeado como GTK, um acrônimo de *"Grammar Tool Kit"*. As seções a seguir apresentam os detalhes do projeto e implantação do GTK. Os resultados produzidos pelo *software* serão expostos na seção 5 deste documento.

4.2 Linguagem GTK

O presente projeto de conclusão de curso foi iniciado criando a linguagem responsável por receber as gramáticas do usuário. As subseções a seguir apresentam os passos para a definição da linguagem proposta.

4.2.1 Padrão adotado

Como toda linguagem de programação, primeiro foi preciso estabelecer os padrões que definem a linguagem. Como resultado, a linguagem foi definida da seguinte forma:

- Todas as variáveis (símbolos não terminais) são denotadas entre '<' e '>'. Por exemplo, para uma variável nomeada de 'A', denota-se '<A>';
- Todos os símbolo terminais presentes na gramática são denotados entre aspas simples. Por exemplo, para escrever o símbolo terminal **if** denota-se 'if';
- O caractere & denota o símbolo de produção vazia (*Lambda*);
- O separador entre regras é denotado pelo caractere | (*pipe*);
- Os caracteres '-' e '>' escritos em sequência representam o símbolo de produção de uma variável;
- Após cada listagem das regras de uma variável, é necessário inserir o caractere ';' para expressar o fim de uma declaração;
- O caractere # é utilizado para realizar comentários no arquivo de entrada, tornando a linha indiferente para o compilador.

A extensão do arquivo foi definida como **.gmr** (uma abreviação de *grammar*). Um exemplo de arquivo de entrada é mostrado na Figura 27.

Figura 27 – Padrão de entreda do *sofware* GTK

```
1 #exemplo: padrão do arquivo de entrada
2 <program>-> <sum> | <div>
3             | <mult> | &;
4 <sum>    -> <number> '+' <number>;
5 <mult>   -> <number> '*' <number>;
6 <div>    -> <number> '/' <number>;
7 <number> -> '1' | '2' | '3' | '4' | '5'
8             | '6' | '7' | '8' | '9';
```

Fonte: Autor

Observando a Figura 27, é possível notar que o compilador aceita quebras de linha na listagem das produções. Após definir o padrão adotado como entrada, foi necessário determinar quais seriam os *tokens* que deveriam ser extraídos da gramática de entrada do usuário, sendo eles:

- 'TK_VARIABLE' : *token* atribuído aos lexemas que representam as variáveis da gramática. Ou seja, as sequências de caracteres escritas entre < e >;
- 'TK_PRODUCTS': *token* atribuído aos lexemas que representam os símbolos de produção presentes na gramática. Ou seja os caracteres '-' e '>' escritos em sequência;
- 'TK_TERMINAL: *token* atribuído aos lexemas que representam os terminais da gramática. Ou seja, as sequências de caracteres escritas entre ' e ' (aspas simples);
- 'TK_LAMBDA': *token* atribuído aos lexemas que representam as produções vazias presentes na gramática. Ou seja, as ocorrências do caractere '&';
- 'TK_PIPE': *token* atribuído aos lexemas que representam os separadores entre regras das variáveis da gramática. Ou seja, o caractere | (*pipe*);
- 'TK_PTVIR': *token* atribuído aos lexemas que representam os fins de comando da gramática.

O próximo passo foi criar a gramática utilizada para a implementação do analisador sintático do compilador. Foi construída uma gramática que atenda o padrão adotado acima, respeitando também as especificações das gramáticas LL(1). Isso se justifica devido ao fato de que o modo de compilação adotado no GTK também segue este método. A Figura 28 apresenta a gramática definida para o processo de compilação do *software*.

Figura 28 – Gramática do compilador GTK

```
< GOAL > → < VAR > ; < REST_GOAL >
< REST_GOAL > → < VAR > 'TK_PTVIR' < REST_GOAL > | λ
< VAR > → 'TK_VARIABLE' 'TK_PRODUCTS' < SEQUENCE >
< SEQUENCE > → < OPTION > < REST_SEQUENCE >
< REST_SEQUENCE > → 'TK_PIPE' < SEQUENCE > | λ
< OPTION > → 'TK_LAMBDA' | < OTHER >
< OTHER > → 'TK_VARIABLE' < REST_OTHER > | 'TK_TEMINAL' < REST_OTHER >
< REST_OTHER > → < OTHER > | λ
```

Fonte: Autor

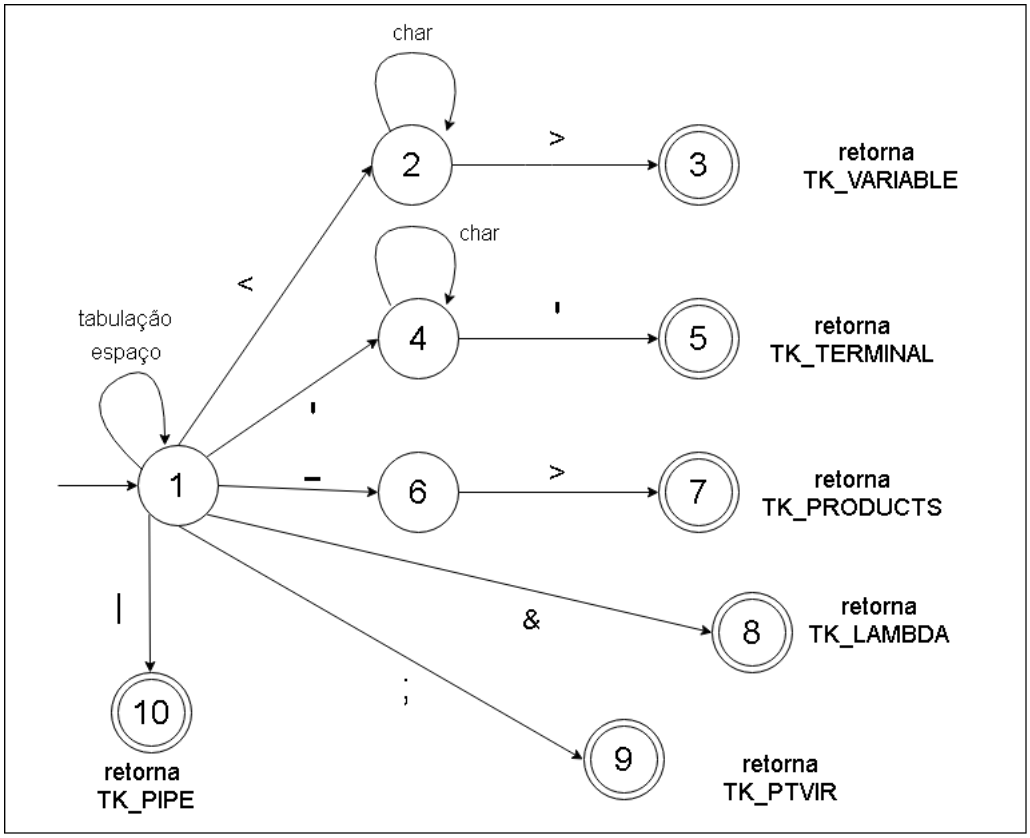
4.3 Criação do compilador para o GTK

Após definir a linguagem, o próximo passo foi a implementação do compilador responsável por ler a entrada do usuário, verificar erros (provendo mensagens em casos positivos) e direcionar a gramática para a estrutura de dados do GTK. As subseções a seguir detalham como todo esse processo de compilação foi realizado.

4.3.1 Analisador léxico do GTK

O primeiro passo na criação do compilador foi a implementação do analisador léxico da linguagem. Foi necessário definir como cada *token* seria extraído do código fonte, sendo assim, foi construído um autômato regular para realizar essa extração. A Figura 29 apresenta o autômato projetado para o analisador léxico do GTK.

Figura 29 – Autômato reconhecedor de *tokens* do GTK



Fonte: Autor

O autômato mostrado na Figura 29 é capaz de extrair todos os lexemas do código fonte de entrada. Após a construção do autômato responsável pela extração dos lexemas do código fonte do usuário, foi realizada a codificação do analisador léxico. Nele está presente as definições dos *tokens* listados acima(subseção 4.2.1), o autômato para extração dos lexemas, a função responsável pelas mensagens de erro e a função *getToken()*.

4.3.2 Analisador sintático GTK

Após a implementação do analisador léxico, a próxima etapa envolveu a criação do analisador sintático. Nessa etapa, foi implementado a estrutura gramatical adotada, recebendo os lexemas disponibilizados pela função *getToken()* do analisador léxico, e mostrando as mensagens de erro ao receber um *token* não esperado pela gramática.

A próxima seção aborda a estrutura do *software* GTK. Nela será mostrado os diagramas de classes presentes nos dois módulos desenvolvidos, além do fluxo que o *software* realiza desde a leitura do arquivo de entrada, até a geração do arquivo HTML de saída.

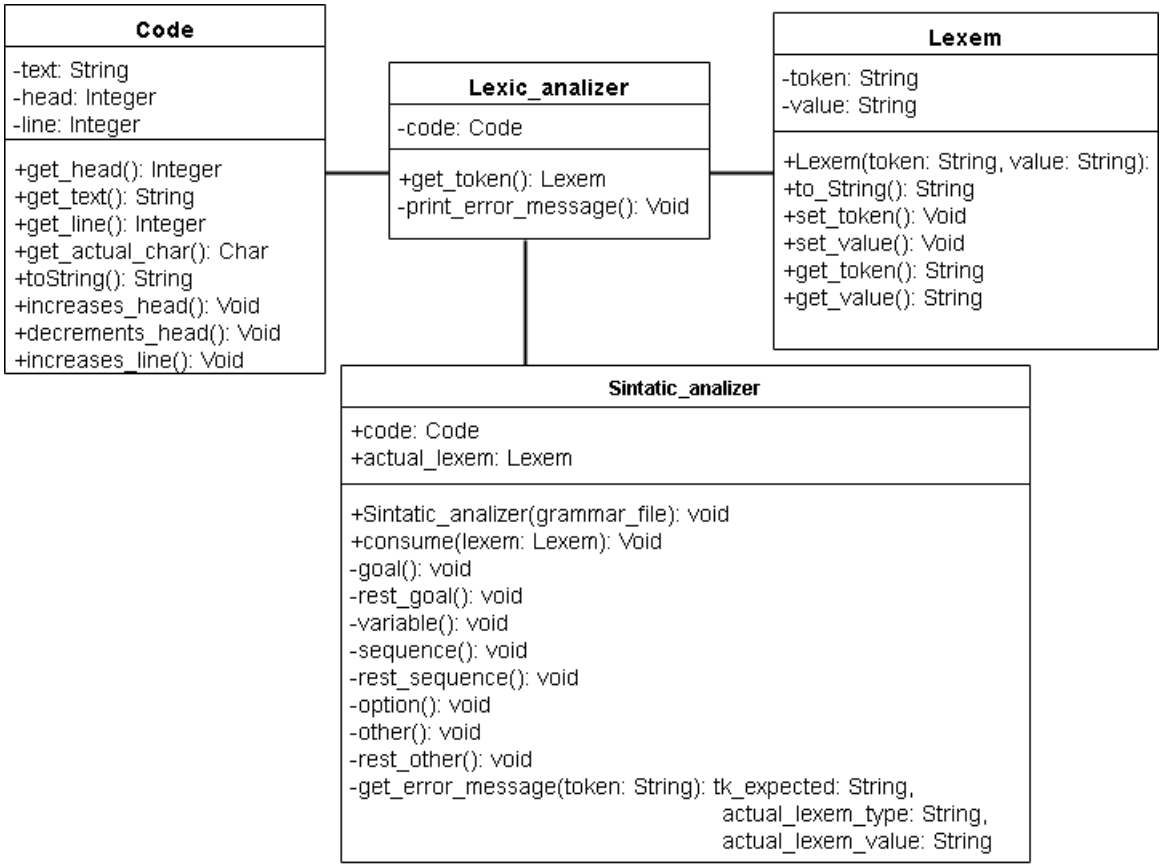
4.4 Estrutura de *software* do GTK

Esta seção tem como objetivo apresentar a estrutura implementada na criação do GTK. O *software* foi separado em quatro módulos: módulo de análise, módulo objeto, módulo de manipulação gramatical e módulo de saída de dados. Os detalhes acerca desses módulos estão presentes nas subseções 4.4.1, 4.4.2, 4.4.3 e 4.4.4 respectivamente, em seguida a seção 4.4.5 apresenta o fluxo do *software*.

4.4.1 Módulo de Análise

O módulo de análise contempla as classes que compõem toda a fase de compilação do código fonte. A Figura 30 apresenta o diagrama de classes que compõem o módulo de análise do GTK, em seguida será discorrido o papel de cada uma dessas classes.

Figura 30 – Diagrama de classes do Módulo de análise do GTK



Fonte: Autor

- Code: A classe code tem como objetivo controlar o código de entrada do usuário. Ela é responsável por avançar e retroceder o ponteiro do caractere atual a ser lido pelo

analisador léxico, e também armazenar a linha atual do ponteiro, útil para exibir o local do problema nos casos de erros sintáticos e léxicos;

- Lexem: representa os lexemas extraídos do código fonte recebido. Ela é composta pelo par ordenado *token-valor* e os métodos de leitura e atualização desses valores;

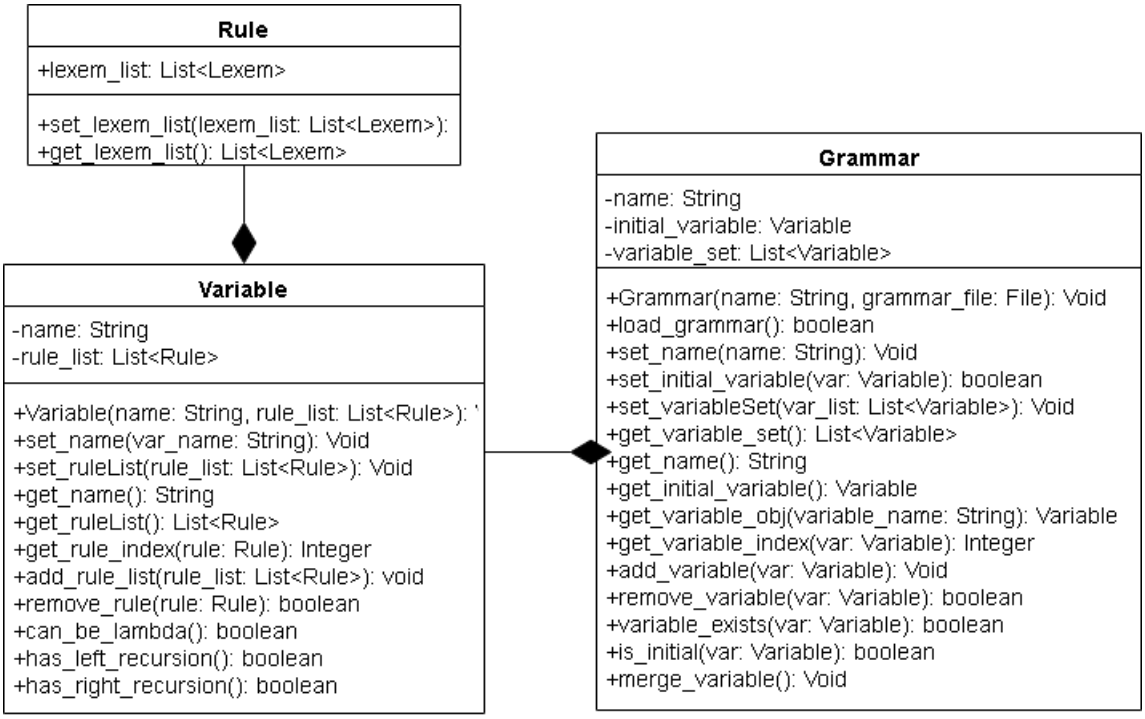
- Lexic_analyzer: é responsável por realizar a análise léxica do GTK. Nela está a função que exibe as mensagens de erros léxicos, além da função *getToken()*, responsável por retornar o próximo lexema lido no código;

- Syntatic_analyzer: é responsável por realizar a análise sintática do GTK. Nela estão os métodos que representam cada variável da gramática do GTK e a exibição das mensagens de erros sintáticos;

4.4.2 Módulo objeto

Após obter sucesso na compilação (módulo de análise), o próximo passo é instanciar a gramática fornecida pelo programa fonte de usuário. Este módulo é responsável pelo instanciamento da gramática de entrada, encaixando suas variáveis e regras nas estruturas de dados implementadas. Estas estruturas foram definidas como mostra a Figura 31.

Figura 31 – Diagrama de classes do Módulo objeto do GTK



Fonte: Autor

- *Grammar*: é a representação computacional uma gramática, possuindo os seguintes atributos: nome, lista de variáveis, e variável inicial da gramática. Esta classe também comporta diversos métodos responsáveis pelo controle da estrutura, como a função que carrega o código de entrada na estrutura, funções de inclusão e remoção de variáveis, verificação de variável inicial e verificação de existência de variáveis;

- *Variable*: representa as variáveis de uma gramática. Possui como atributos o nome da variável e uma lista de regras. Também comporta os métodos de adição, atualização e remoção de regras, além de funções que verificam a existência de recursão na mesma;

- *Rule*: representa as regras de uma variável. Uma regra é composta por uma sequência de lexemas.

Como mostra a Figura 31, as três classes possuem uma relação entre si. A classe *Grammar* possui uma relação de composição com a classe *Variable*, mostrando que para uma gramática existir, então é necessário que esta possua uma ou mais variáveis que a compõem. A classe *Variable* por sua vez, possui uma relação de composição com a classe *Rule*. Sendo assim para uma variável existir, esta precisa conter uma ou mais regras em sua composição.

4.4.3 Módulo de Manipulação Gramatical

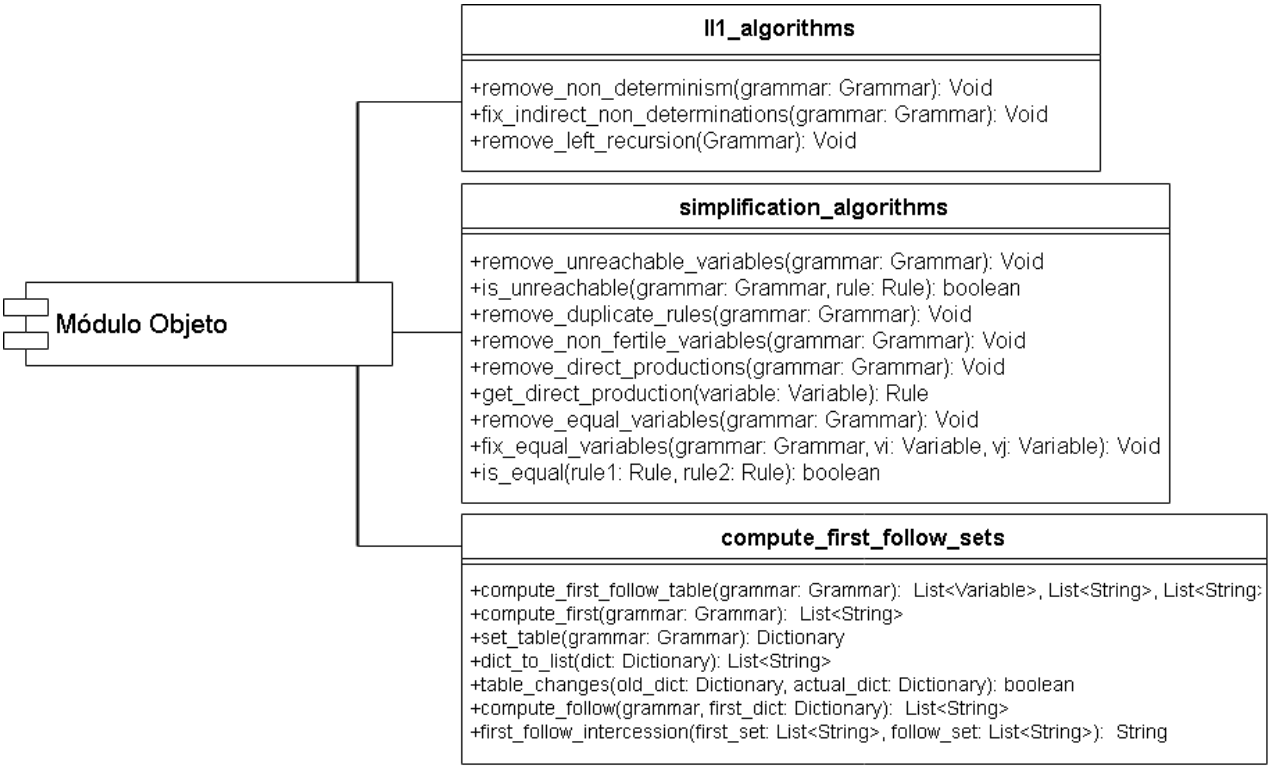
Obtendo sucesso na compilação (Módulo de Análise) e ser instanciada (Módulo Objeto), a gramática expressada pelo programa fonte do usuário está apta a ser utilizada nas manipulações disponibilizadas pelo GTK. O Módulo de Manipulação Gramatical comporta as ferramentas que realização as alterações nassa gramática. A Figura 32 apresenta o esquemático adotado para a sua implementação, em seguida será mostrado os objetivos de cada biblioteca disponível neste módulo.

- *ll1_algorithms*: é a biblioteca que possui as funções que alteram a gramática para o formato LL(1). Esta possui as funções de remoção de não-determinismo e remoção de recursão à esquerda;

- *simplification_algorithms*: biblioteca que comporta as funções de simplificação gramatical. Esta possui as funções de remover variáveis inalcançáveis, remoção de regras duplicadas, remoção de variáveis inférteis, remoção de produções diretas e remoção de variáveis idênticas;

- *compute_first_follow_sets*: esta biblioteca possui as funções necessárias para o cálculo dos conjuntos *First* e *Follow*.

Figura 32 – Diagrama de classes do Módulo de manipulação do GTK

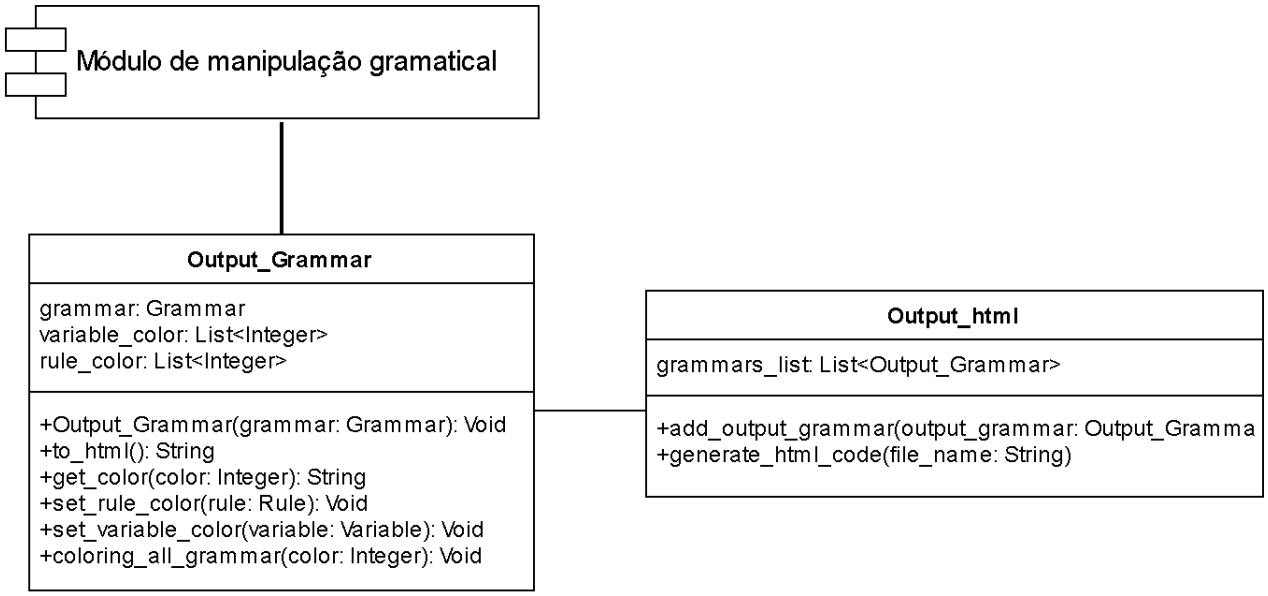


Fonte: Autor

4.4.4 Módulo de Saída de dados

O módulo de saída de dados é responsável por exibir ao usuário o passo a passo realizado nas funções disponibilizadas pelo módulo de manipulação gramatical (ver subseção 4.4.3), sendo assim, o mesmo é utilizado pelos seus algoritmos. A Figura 33 ilustra a estrutura deste módulo de saída de dados.

Figura 33 – Diagrama de classes do Módulo de saída de dados do GTK



Fonte: Autor

A cada modificação relevante na gramática de entrada, realizada pelas funções de manipulação (ver subseção 4.4.3), o módulo de saída de dados instancia um objeto da classe *Output_Grammar*, que representa estado atual da gramática. Após instanciar, este estado da gramática é inserido na lista de gramáticas *grammars_list* da classe *Output_html*, através do método *add_output_grammar()*. No fim de todas as manipulações realizadas, a classe *Output_html* terá armazenado o passo-a-passo das manipulações gramaticais. Por fim o método *generate_html_code()*, da classe *Output_html*, gera uma página HTML com todo o processo de manipulação gramatical.

Os tópicos a seguir detalham os objetivos das duas classes que compõem este módulo.

- *Output_Grammar*: Esta classe possui como atributos um objeto Grammar e duas listas, *variable_color* e *rule_color*. O atributo *variable_color* tem como objetivo armazenar a cor de cada variável da gramática. O atributo *rule_color* por sua vez, controla as cores de cada regra da gramática. Essas cores serão utilizadas no momento da escrita da página HTML, para buscar uma melhor visualização das

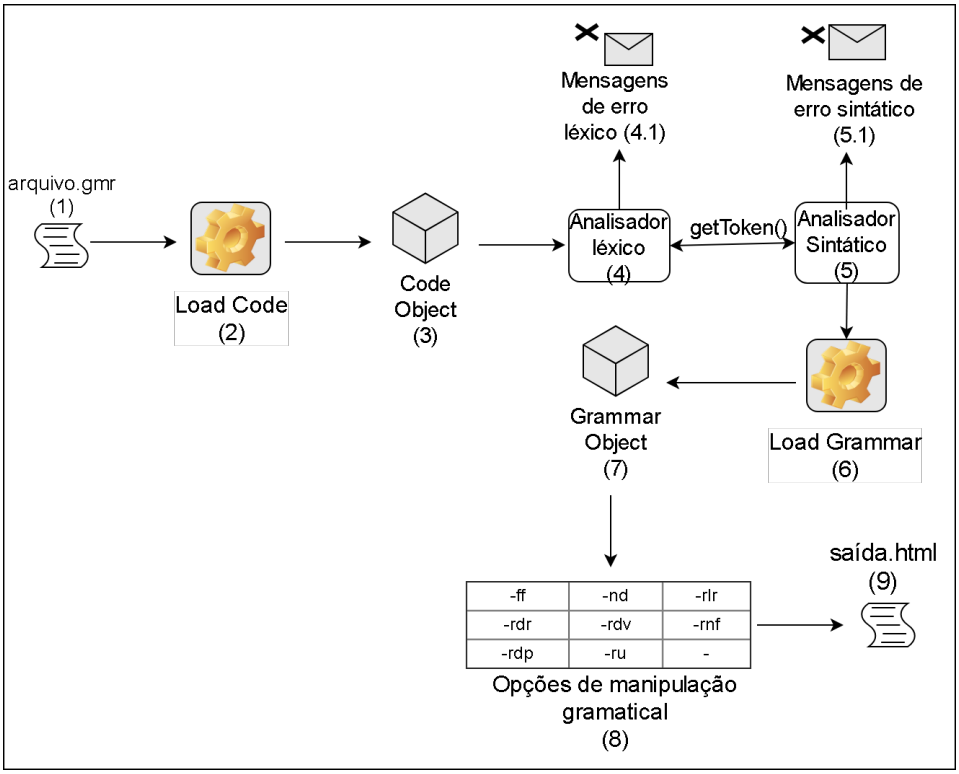
manipulações realizadas. Como são utilizadas em uma página HTML, o valor que representa essas cores é expressado no formato hexadecimal. Esta classe também possui os métodos de coloração das regras e variáveis, além do método *to_html()*, responsável por converter o objeto *Output_Grammar* para o formato HTML;

- *Output_html*: Esta classe é responsável por armazenar a lista com os objetos *Output_Grammar*, que serão mostradas ao usuário na página HTML com o passo-a-passo das manipulações. A classe disponibiliza os métodos de inserção na lista de gramáticas (*grammars_list*), geração da página HTML que exibe o passo-a-passo realizado.

4.4.5 Fluxo do *software*

Após a apresentação dos dos módulos que compõem o *Grammar Tool Kit*, esta seção tem como objetivo apresentar o percurso realizado no *software*, desde a gramática de entrada do usuário, até a saída em formato HTML. A Figura 34 ilustra o fluxo do *software* GTK.

Figura 34 – Fluxo do *software*



Fonte: Autor

- Este item representa a entrada de extensão **.grm** fornecida pelo usuário;

- Nesta etapa o arquivo fornecido pelo usuário é processado a fim de transformá-lo em um objeto da classe Code;
- Este item representa o código instanciado na estrutura *Code*, estando apto a ser compilado;
- A etapa 4 trata do analisador léxico do GTK, que fornece a função *getToken()* para a extração dos lexemas do código de entrada. Esta etapa também é responsável por emitir mensagens de erros léxicos, apresentando uma mensagem de erro e interrompendo o processo de compilação;
- Representa o analisador sintático do GTK, esta etapa é responsável por usar a função *getToken()* fornecida pelo analisador léxico (4) e aplicar o lexemas recebidos na estrutura gramatical do *software*. Esta etapa também é responsável por emitir mensagens de erros sintáticos, interrompendo o processo de compilação quando o mesmo ocorrer;
- Após obter sucesso na compilação realizada nas etapas anteriores, a etapa 6 tem como objetivo carregar o código instanciado (3) para a estrutura *Grammar*;
- Representa o código usuário carregado para a estrutura *Grammar* instanciada, a partir desta etapa, a gramática de entrada do usuário está pronta para ser utilizada nos algoritmos de manipulação gramatical (8);
- Esta etapa tem como objetivo realizar a manipulação requisitada pelo usuário, de acordo com os parâmetros de entrada disponíveis (mostrados na seção 4.5 deste documento);
- Após a manipulação realizada na etapa 8, a etepa 9 gera o HTML de saída, apresentando o passo a passo realizado de acordo com o parâmetro de entrada.

4.5 Ferramentas e Parâmetros de Entrada do Usuário

Assim como várias ferramentas desenvolvidas para a plataforma GNU/Linux, o GTK é um *software* que apresenta o terminal como interface de usuário. A seguir será mostrado as ferramentas que o *Grammar Tool Kit* disponibiliza ao usuário e os parâmetros usados para o uso de cada uma das funcionalidades.

- **-ff** : comando utilizado para calcular e gerar a tabela *First* & *Follow* de uma gramática. Exemplo de uso: *./gtk -ff gramatica.gmr*
- **-nd**: comando utilizado para remover todas as indeterminações da gramática. Exemplo de uso: *./gtk -nd gramatica.gmr*

- **-ru**: comando utilizado para remover as variáveis inalcançáveis da gramática. *./gtk -ru gramatica.gmr*
- **-rdv**: comando utilizado para modificar a gramática de forma que não possua variáveis duplicadas. Exemplo de uso *./gtk -rdv gramatica.gmr*
- **-rnf**: comando utilizado para remover as variáveis inférteis da gramática. Exemplo de uso: *./gtk -rnf gramatica.gmr*
- **-rdp**: comando utilizado para modificar a gramática de forma que não possua mais produções diretas em suas variáveis. Exemplo de uso *./gtk -rdp gramatica.gmr*
- **-rlr**: comando utilizado para transformar a gramática, gerando uma gramática equivalente sem nenhuma recursão a esquerda em suas variáveis. Exemplo de uso: *./gtk -rlr gramatica.gmr*

4.6 Experimentos Realizados

Para a validação do *software* foram elaborados quatro cenários, com objetivo de abordar todos os algoritmos implementados no GTK. Para o Cenário I, foi criada uma gramática que possui todos os problemas passíveis de simplificação(abordados na seção 2.8). Para os Cenários II e III foram selecionadas gramáticas presentes nas literaturas que foram pesquisadas para a realização do trabalho. Por fim, para o Cenário IV foi escolhida uma gramática que foi utilizada em um trabalho prático da disciplina de compiladores. As subseções a seguir apresentam os detalhes sobre cada um dos seis cenários.

4.6.1 Cenário I

O primeiro cenário tem como objetivo abordar os algoritmos de simplificação. Neste experimento foi criada uma gramática passível de ser simplificada por todos os algoritmos presentes na seção 2.8.

Figura 35 – Gramática para experimento de simplificação - Cenário I

$$
\begin{array}{l}
<S*> \rightarrow a <S> b \mid <M> \\
<M> \rightarrow c <M> d \mid c d \mid c <M> d \mid <N> \\
<C> \rightarrow c c <C> \mid d d <D> \\
<D> \rightarrow d <D> \mid d \\
<B> \rightarrow c <M> d \mid c d \mid c <M> d \\
<N> \rightarrow c <N> d \mid <N> d \mid c <N>
\end{array}
$$

Fonte: Autor

A gramática expressa pela Figura 35, gera a linguagem livre de contexto $L = \{w|w \in a^i c^j d^j b^i | i >= 0 | j > 0\}$. É esperado que como resultado, os algoritmos sejam capazes de auxiliar na simplificação da gramática, gerando a mesma linguagem, mas com um número reduzido de variáveis e produções.

O experimento do Cenário I foi realizado da seguinte forma: dada como entrada a gramática representada pela Figura 35, a mesma será entrada para o primeiro dos algoritmos de simplificação. Logo depois, a gramática resultante será entrada para o próximo algoritmo, seguindo este processo até que a gramática passe por todos os algoritmos de simplificação disponíveis no GTK. Este processo seguirá a seguinte ordem.

1. Primeiro, a gramática de entrada será modificada utilizando o parâmetro **-rnf**, responsável por remover as variáveis inférteis da gramática;

2. Em seguida, a gramática resultante do processo do item (1) será modificada utilizando o parâmetro **-rdr**, responsável por remover as regras duplicadas de uma mesma variável;

3. O próximo algoritmo, aplicado a gramática resultante do item (2), será o de remoção de variáveis duplicadas(variáveis idênticas), utilizando o parâmetro **-rdv**;

4. O último algoritmo a ser executado sobre a gramática resultante do item (3), será o de remoção de variáveis inalcançáveis. Para isso será utilizado o parâmetro **-ru**.

Após a aplicação dos métodos de simplificação citados, é esperado que a gramática esteja simplificada, não apresentando variáveis inférteis, regras duplicadas, variáveis duplicadas e variáveis inalcançáveis.

4.6.2 Cenário II

No segundo cenário também serão utilizados os algoritmos de simplificação. Mas nesse caso, a entrada utilizada será uma gramática presente no livro *'Introdução aos fundamentos da Computação'* de Vieira (2006). Em seu livro, esta gramática foi utilizada para exemplificar o uso de dois algoritmos de simplificação, que também serão aplicados neste cenário.

A gramática representada pela Figura 36, apresenta duas possíveis possibilidades de simplificação: remoção de variáveis inférteis e remoção de variáveis inalcançáveis.

Figura 36 – Gramática para experimento de simplificação - Cenário II

```
< A > → < A > < B > < C > | < A > < E > < F > | < B > < D >
< B > → < B > 0 | 0
< C > → 0 < C > | < E > < B >
< D > → 1 < D > | 1
< E > → < B > < E >
< F > → 1 < F > 1 | 1
```

Fonte: Autor

Após a aplicação dos dois algoritmos citados, é esperado que como saída, o GTK apresente uma gramática similar a resolução de Vieira (2006). Ou seja, não necessariamente idêntica, mas livre dos problemas de simplificação que os dois algoritmos têm o objetivo de resolver. Além disso, é esperado que a gramática gerada pelo GTK, expresse a mesma linguagem que a gramática resultante de Vieira (2006).

4.6.3 Cenário III

O Cenário três foi definido da seguinte maneira. Márcio Eduardo Delamaro define a gramática da Figura 37 como um caso clássico de recursividade à esquerda. Em seu livro 'Como construir um compilador utilizando ferramentas Java', Delamaro (2004) apresenta esta gramática e realiza dois procedimentos: remoção de recursão à esquerda e Fatoração à esquerda. O intuito do autor é gerar uma gramática equivalente, mas que se adeque ao formato LL(1). De forma semelhante, este cenário tem como objetivo receber a gramática denotada pela literatura, e realizar os mesmos procedimentos. Como consequência, é esperado que a gramática resultante, gerada pelo GTK, produza uma saída equivalente ao resultado obtido por Delamaro (2004).

Figura 37 – Gramática para experimento LL(1) - Cenário III

```
< expression > → < expression > + < term > | < expression > − < term > | < term >
< term > → < term > * < factor > | < term > / < factor >
< factor > → id | const | ( < expression > )
```

Fonte: Autor

4.6.4 Cenário IV

No segundo semestre de 2018, os alunos de compiladores do Bacharelado em Ciência da Computação do IFMG - Campus Formiga, realizaram como trabalho da disciplina a construção dos analisadores léxico e sintático para a linguagem de programação *MiniC*.

Para a implementação desses analisadores, foi necessário o cálculo dos conjuntos FF relativos a gramática que expressa a linguagem. Para isto, foi utilizado como ferramenta para o cálculo, o *software* descrito por este trabalho. A Figura 38 apresenta a gramática construída para a implementação da linguagem *MiniC*.

Figura 38 – Gramática Mini C - Cenário IV

```
<function*> -> <type> 'IDENT' '(' <argList> ')' <bloco> ;
<argList> -> <arg> <restoArgList> | & ;
<arg> -> <type> 'IDENT' ;
<restoArgList> -> ',' <argList> | & ;
<type> -> 'int' | 'float' ;
<bloco> -> '{' <stmtList> '}' ;
<stmtList> -> <stmt> <stmtList> | & ;
<stmt> -> <forStmt> | <ioStmt> | <whileStmt> | <expr> ';' | <ifStmt>
  | | <bloco>
    | <declaration>
    | ';' ;
<declaration> -> <type> <identList> ';' ;
<identList> -> 'IDENT' <restoIdentList> ;
<restoIdentList> -> ',' 'IDENT' <restoIdentList> | & ;
<forStmt> -> 'for' '(' <optExpr> ';' <optExpr> ';' <optExpr> ')' <stmt> ;
<optExpr> -> <expr> | & ;
<ioStmt> -> 'scan' '(' 'IDENT' ')' ';' | 'print' '(' <outList> ')' ';' ;
<outList> -> <out> <restoOutList> ;
<out> -> 'STR' | 'IDENT' | 'NUMint' | 'NUMfloat' ;
<restoOutList> -> ',' <out> <restoOutList> | & ;
<whileStmt> -> 'while' '(' <expr> ')' <stmt> ;
<ifStmt> -> 'if' '(' <expr> ')' <stmt> <elsePart> ;
<elsePart> -> 'else' <stmt> | & ;
<expr> -> <atrib> ;
<atrib> -> <or> <restoAtrib> ;
<restoAtrib> -> '=' <atrib> | & ;
<or> -> <and> <restoOr> ;
<restoOr> -> '||' <and> <restoOr> | & ;
<and> -> <not> <restoAnd> ;
<restoAnd> -> '&&' <not> <restoAnd> | & ;
<not> -> '!' <not> | <rel> ;
<rel> -> <add> <restoRel> ;
<restoRel> -> '==' <add> | '!=' <add>
            | '<' <add> | '<=' <add>
            | '>' <add> | '>=' <add> | & ;
<add> -> <mult> <restoAdd> ;
<restoAdd> -> '+' <mult> <restoAdd>
            | '-' <mult> <restoAdd> | & ;
<mult> -> <uno> <restoMult> ;
<restoMult> -> '*' <uno> <restoMult>
            |  '/' <uno> <restoMult>
            |  '%' <uno> <restoMult> | & ;
<uno> -> '+' <uno> | '-' <uno> | <fator> ;
<fator> -> 'NUMint' | 'NUMfloat'
         | 'IDENT'  | '(' <atrib> ')' ;
```

Fonte: Autor

A partir destas informações, o Cenário IV tem como objetivo apresentar o cálculo dos conjuntos *First* e *Follow* para a gramática desenvolvida em sala de aula.

5 RESULTADOS E ANÁLISE

Nesta seção serão apresentados todos os resultados obtidos por todos os cenários propostos na subseção anterior (4.6), bem como análises dos resultados obtidos por eles.

5.1 Cenário I

Como mostrado na subseção 4.6, o Cenário I tem como objetivo verificar os algoritmos de simplificação disponíveis no GTK. Para isto, os algoritmos serão aplicados na gramática apresentada.

De acordo com a ordem de aplicação dos algoritmos que foram utilizados no Cenário I, o primeiro passo é remover as variáveis inférteis. Observando a gramática expressa pela Figura 35 é possível notar que a variável $< N >$ é infértil, pois apresenta recursividade em todas as suas derivações. Após a aplicação do algoritmo sobre a gramática inicial, o GTK obteve a gramática apresentada pela Figura 39, removendo a variável $< N >$.

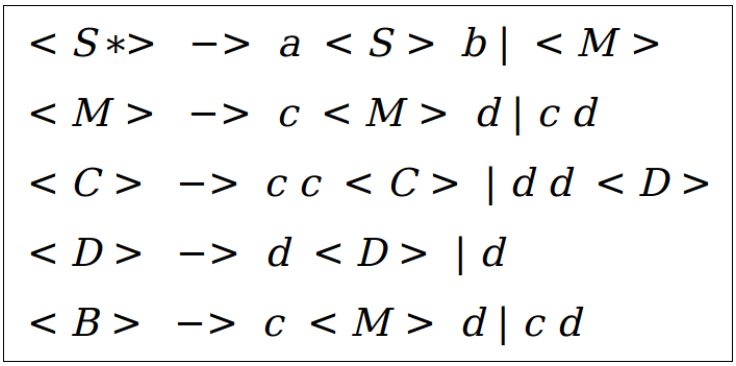
Figura 39 – Remção de $< N >$ - Cenário I

$$
\begin{aligned}
&< S* > \ \rightarrow \ a \ < S > \ b \mid < M > \\
&< M > \ \rightarrow \ c \ < M > \ d \mid c \ d \mid c \ < M > \ d \\
&< C > \ \rightarrow \ c \ c \ < C > \mid d \ d \ < D > \\
&< D > \ \rightarrow \ d \ < D > \mid d \\
&< B > \ \rightarrow \ c \ < M > \ d \mid c \ d \mid c \ < M > \ d
\end{aligned}
$$

Fonte: Autor

Como mostrado na subseção 4.6.1, o próximo passo foi aplicar o algoritmo de remoção de regras duplicadas. Nota-se na Figura 39, na variável $< M >$ existem duas ocorrências da regra $< M > \Rightarrow c < M >$. De forma semelhante, na variável $< B >$ existem duas ocorrências de $< B > \Rightarrow c < M > d$. Sendo assim, é possível remover uma das ocorrências em cada umas dessas variáveis. A Figura 40 apresenta a gramática resultante após a aplicação do algoritmo de remoção de regras duplicadas. Após a aplicação, é possível notar a ausência da duplicação das regras.

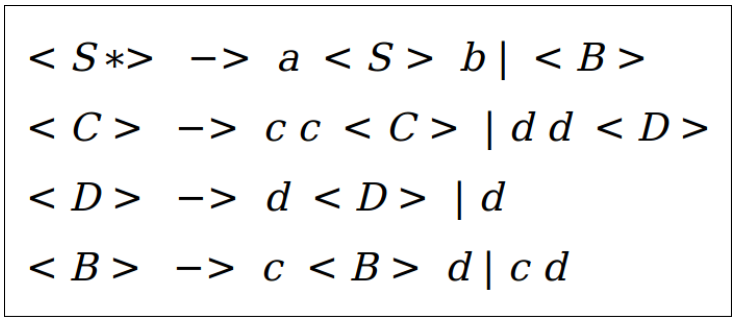
Figura 40 – Remoção das regras duplicadas - Cenário I



Fonte: Autor

Em seguida, foi aplicado sobre a gramática resultante (Figura 40) o algoritmo de remoção de variáveis duplicadas. Observando a Figura 40, é possível ver que as variáveis $< B >$ e $< M >$ possuem as mesmas derivações (são idênticas), sendo assim, uma das duas é desnecessária, pode-se então excluir a variável $< M >$ e substituir as ocorrências da mesma, por $< B >$. Sendo assim, o resultado obtido pelo GTK foi a gramática apresentada pela Figura 41.

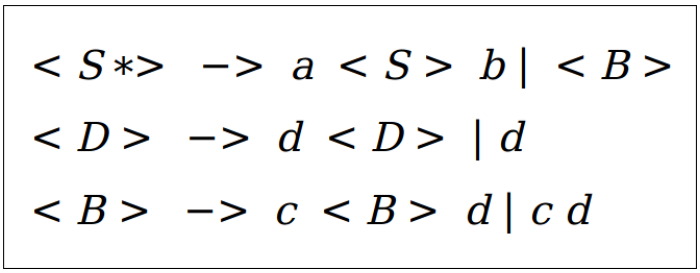
Figura 41 – Remoção das variáveis duplicadas - Cenário I



Fonte: Autor

O último algoritmo a ser aplicado é a remoção de variáveis inalcançáveis, assunto abordado na subseção 3.2.2.6. Como pode ser observado na Figura 41, a variável $< C >$ não aparece em nenhuma derivação de nenhuma outra regra. Como consequência, a esta variável não é alcançável a partir da variável inicial $< S >$, podendo ser eliminada sem alterar a linguagem produzida pela gramática. A Figura 42 mostra o resultado encontrado após o GTK executar o algoritmo sobre a gramática expressada pela Figura 41.

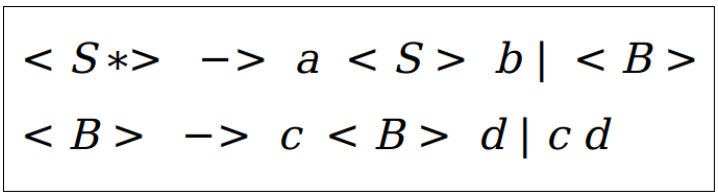
Figura 42 – Remoção das variáveis inalcançáveis - Cenário I



Fonte: Autor

A gramática resultante (Figura 42) já não apresenta a variável $< C >$. O último passo realizado no Cenário I, é novamente a execução do algoritmo de remoção de variáveis inférteis. É possível notar que com a remoção de $< C >$, a variável $< D >$ também se mostrou infértil. Então, de forma semelhante, $< D >$ também pode ser removida. A Figura 43 mostra o resultado final após a aplicação dos algoritmos de simplificação propostos pelo Cenário I.

Figura 43 – Gramática final - Cenário I



Fonte: Autor

Como resultado, obteve-se uma gramática (Figura 43) que expressa a mesma linguagem da gramática denotada pela gramática inicial (Figura 35).

5.2 Cenário II

Dada a especificação do Cenário II na subseção 4.6.2, primeiro foi executado por Veira (2006) o algoritmo de remoção de variáveis inférteis sobre a gramática inicial expressada pela Figura 36. Nela é possível observar que a variável $< E >$ possui apenas uma opção de derivação, e esta se mostra recursiva. Sendo assim, a variável $< E >$ se mostra infértil, podendo ser excluída da gramática. Utilizando parâmetro **-rnf**, a saída obtida pelo *software* GTK é mostrada pela Figura 44. Nota-se que, além de excluir a

variável $< E >$ (que era infértil), também foram retiradas da gramática inicial (Figura 36) todas as regras que apresentavam $< E >$ como possível derivação.

Figura 44 – Remoção da variável infértil $< E >$ - Cenário II

<A> -> <A> <B> <C> | <B> <D>
<B> -> <B> 0 | 0
<C> -> 0 <C>
<D> -> 1 <D> | 1
<F> -> 1 <F> 1 | 1

Fonte: Autor

Ao remover $< E >$ do conjunto de variáveis da gramática, $< C >$ também se mostrou infértil, por possuir recursão em sua única derivação possível. Então, novamente foi aplicado o algoritmo de remoção de variáveis inférteis, obtendo a gramática da Figura 45 como resultado.

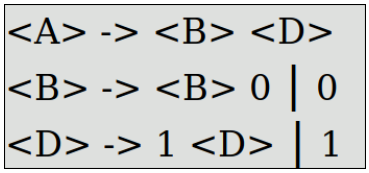
Figura 45 – Remoção da variável infértil $< C >$ - Cenário II

<A> -> <B> <D>
<B> -> <B> 0 | 0
<D> -> 1 <D> | 1
<F> -> 1 <F> 1 | 1

Fonte: Autor

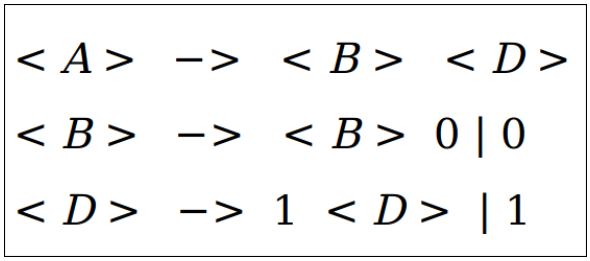
Após a remoção das variáveis inférteis, Vieira (2006) aplica o algoritmo de remoção de variáveis inalcançáveis. Seguindo os passos da literatura, foi executado o mesmo algoritmo, utilizando o GTK. O resultado obtido pelo *software* é mostrado pela Figura 46, e se mostrou idêntico ao resultado final obtido por Vieira (2006) (presente na Figura 47). Sendo assim, foi considerado que os experimentos realizados no Cenário II obtiveram sucesso.

Figura 46 – Resultado GTK - Cenário II



Fonte: Autor

Figura 47 – Resultado da literatura - Cenário II



Fonte: Autor

5.3 Cenário III

Como foi proposto pela subseção 4.6.3, o Cenário III tem como objetivo realizar o procedimento de transformação gramatical feito por Delamaro (2004). Dada a gramática inicial mostrada pela Figura 37, o objetivo deste cenário é fazer com que a mesma se encaixe nos padrões LL(1), assim como fez Delamaro (2004) em seu livro.

A gramática em questão (Figura 37) possui recursividade à esquerda nas variáveis $< expression >$ e $< term >$. O primeiro passo a ser realizado é a remoção dessas recursividades. Para isso, assim como fez Delamaro (2004), foi executado o algoritmo de remoção de recursividade à esquerda, mas utilizando o GTK através do parâmetro **-rlr**. Como resultado, foi obtida a gramática expressada pela Figura 48.

Figura 48 – Remoção de recursividade à esquerda GTK - Cenário III

```
< expression > -> < term > | < term > < expression' >
< term > -> < factor > | < factor > < term' >
< factor > -> id | const | ( < expression > )
< expression' > -> + < term > |
                   + < term > < expression' > | - < term > | - < term > < expression' >
< term' > -> * < factor > | * < factor > < term' > | / < factor > | / < factor > < term' >
```

Fonte: Autor

Como é possível observar na Figura 48, o primeiro objetivo foi alcançado. A gramática resultante não apresenta mais recursividade à esquerda em nenhuma de suas variáveis, obtendo assim o mesmo resultado alcançado pela literatura. Seguindo os passos realizados por Delamaro (2004), em seguida foi executada a remoção de indeterminações na gramática (fatoração à esquerda). Utilizando o parâmetro **-nd**, a saída obtida pelo GTK é a gramática mostrada pela Figura 48. Neste experimento foi notado uma gramática diferente da encontrada pela literatura. Apesar de diferentes, ambas podem ser equivalentes, ou seja, gerar a mesma linguagem. A gramática resultante de Delamaro (2004) está presente na Figura 49.

Figura 49 – Resultado fatoração à esquerda GTK - Cenário III

```
< expression > -> < term > < expression'' >
< expression' > -> + < term > < expression''' > | - < term > < expression''' >
< expression'' > -> λ | < expression' >
< expression''' > -> < term > < expression'''' >
< expression'''' > -> λ | < expression' >
< term > -> < factor > < term'' >
< term' > -> * < factor > < term''' > | < factor > < term''' >
< term'' > -> λ | < term' >
< term''' > -> < factor > < term'''' >
< term'''' > -> λ | < term' >
< factor > -> id | const | ( < expression > )
```

Fonte: Autor

Figura 50 – Resultado fatoração à esquerda literatura - Cenário III

```
expression → term expression'
expression' → - term expression' | + term expression' | λ
term → fator term'
term' → * fator term' | / fator term' | λ
fator → id | const | ( expression )
```

Fonte: Delamaro (2004)

O algoritmo de equivalência entre gramáticas é um problema reconhecidamente indecidível pela computação (Aho et. al. 2008). Então como tentativa de verificar que as gramáticas são equivalentes, foram utilizados alguns algoritmos de simplificação disponíveis no GTK e reorganizações manuais das regras presentes na gramática.

Tendo como inicial a gramática denotada pela Figura 49, o primeiro passo foi substituir as duas ocorrências da variável <expression"'> (presentes nas regras $< expression' > \Rightarrow + < expression''' >$ e $< expression' > \Rightarrow - < expression''' >$), pela única regra que ela possui ($< expression''' > \Rightarrow < term >< expression'''' >$). De forma semelhante, o mesmo foi realizado com a variável $< term''' >$, substituindo sua única produção (<term"'> -> <factor> <term""'>) nas produções onde ela aparece ($< term' > \Rightarrow * < term''' >$ e $< term' > \Rightarrow / < term''' >$). Ao realizar essas duas modificações, a gramática resultante é a mostrada pela Figura 51.

Figura 51 – Modificações manuais realizadas - Cenário III

< *expression* > → < *term* > < *expression*″ >
< *expression*′ > → + < *term* > < *expression*‴ > | − < *term* > < *expression*‴ >
< *expression*″ > → λ | < *expression*′ >
< *expression*‴ > → < *term* > < *expression*‴ >
< *expression*‴ > → λ | < *expression*′ >
< *term* > → < *factor* > < *term*″ >
< *term*′ > → ∗ < *factor* > < *term*‴ > | / < *factor* > < *term*‴ >
< *term*″ > → λ | < *term*′ >
< *term*‴ > → < *factor* > < *term*‴ >
< *term*‴ > → λ | < *term*′ >
< *factor* > → *id* | *const* | (< *expression* >)

Fonte: Autor

Em seguida, como é possível ver na Figura 51, as variáveis $< term''' >$ e $< expression''' >$ se mostraram inúteis. Sendo assim, foi executado o algoritmo para a remoção destas variáveis, tendo como resultado a gramática mostrada pela Figura 52.

Figura 52 – Remoção de variáveis inúteis - Cenário III

< *expression* > → < *term* > < *expression*″ >
< *expression*′ > → + < *term* > < *expression*‴ > | − < *term* > < *expression*‴ >
< *expression*″ > → λ | < *expression*′ >
< *expression*‴ > → λ | < *expression*′ >
< *term* > → < *factor* > < *term*″ >
< *term*′ > → ∗ < *factor* > < *term*‴ > | / < *factor* > < *term*‴ >
< *term*″ > → λ | < *term*′ >
< *term*‴ > → λ | < *term*′ >
< *factor* > → *id* | *const* | (< *expression* >)

Fonte: Autor

Após a remoção das variáveis inúteis, foi notado duas ocorrências de variáveis duplicadas(idênticas). Observando a Figura 52, é possível notar dois pares desse problema, são eles:

- $< expression'' >$ e $< expression'''' >$;
- $< term'' >$ e $< term'''' >$.

Executando o algoritmo de remoção de variáveis duplicadas, foi obtida a gramática denotada pela Figura 53.

Figura 53 – Remoção de variáveis duplicadas - Cenário III

```
< expression > -> < term > < expression'' >
< expression' > -> + < term > < expression'' > | - < term > < expression'' >
< expression'' > -> λ | < expression' >
< term > -> < factor > < term'' >
< term' > -> * < factor > < term'' > | / < factor > < term'' >
< term'' > -> λ | < term' >
< factor > -> id | const | ( < expression > )
```

Fonte: Autor

Realizando a operação acima, a gramática resultante (Figura 53) se mostrou mais próxima à gramática da literatura, mas ainda é passível de duas simplificações. A primeira é a remoção de produções diretas. Como pode ser observado na Figura 53, existe a ocorrência de duas produções diretas na gramática, uma na regra $< expression'' > \Rightarrow < expression' >$ e outra em $< term'' > \Rightarrow < term' >$. Utilizando o algoritmos de remoção de produções diretas, através do parâmetro **-rdp**, foi obtida a gramática mostrada pela Figura 54.

Figura 54 – Remoção de produções diretas - Cenário III

```
< expression > -> < term > < expression'' >
< expression' > -> + < term > < expression'' > | - < term > < expression'' >
< expression'' > -> λ | < expression' >
< term > -> < factor > < term'' >
< term' > -> * < factor > < term'' > | / < factor > < term'' >
< term'' > -> λ | < term' >
< factor > -> id | const | ( < expression > )
```

Fonte: Autor

Por fim, a gramática resultante (Figura 54) pode ser simplificada uma última vez. Aos remover as produções diretas, as variáveis $< expression' >$ e $< term' >$ não são utilizadas por nenhuma outra variável, portanto são inúteis na gramática. Sendo assim,

puderam ser eliminadas pelo algoritmo de remoção de variáveis inúteis, resultando na gramática presente na Figura 55.

Figura 55 – Resultado - Cenário III

```
< expression > -> < term > < expression'' >
< expression'' > -> λ | + < term > < expression'' > | - < term > < expression'' >
< term > -> < factor > < term'' >
< term'' > -> λ | * < factor > < term'' > | / < factor > < term'' >
< factor > -> id | const | ( < expression > )
```

Fonte: Autor

Como é possível notar, a gramática presente na Figura 55 é similar a gramática encontrada por Delamaro (2004)(Figura 50). Portanto foi considerado que os experimentos realizados no Cenário III obtiveram sucesso.

5.4 Cenário IV

Como dito na seção 4.6.4, o Cenário IV tem como objetivo apresentar o resultado obtido para a gramática do *MiniC* (Figura 38). Após executar o *software* para a gramática, os resultados obtidos estão presentes na tabela presente na Figura 56.

De acordo com a tabela resultante da saída do GTK, aprsentada na Figura 56, a coluna *Variable* denota cada variável presente na gramática; a coluna *First Set* representa os todos terminais que podem aparecer primeiro, ao derivar uma variável; já a coluna *Follow Set* apresenta todos os terminais que podem aparecer após derivar uma variável.

Como é possível observar, logo abaixo da tabela o *software* levanta um alerta. Esta mensagem apresenta um problema abordado na seção 2.7.2 deste documento. O alerta levantado aponta que a variável $< elsePart >$, que pode derivar em uma cadeia vazia, possui uma intersecção entre os seus conjuntos *First* e *Follow*. Como dito anteriormente (seção x.x.x), este é um problema recorrente e geralmente é solucionado nas seguintes fases do processo de compilação. Portanto, apesar de apresentar esse problema, a gramática de entrada está apta a ser implementada nos analisadores léxico e sintático do compilador.

Figura 56 – Cálculo dos conjuntos *First* e *Follow* MiniC - Cenário IV

Variable	First Set	Follow Set
<function>	float int	$
<argList>	λ float int	)
<arg>	float int	) ,
<restoArgList>	λ ,	)
<type>	float int	IDENT
<bloco>	{	! $ (+ - ; IDENT NUMfloat NUMint else float for if int print scan while { }
<stmtList>	! λ (+ - ; IDENT NUMfloat NUMint float for if int print scan while {	}
<stmt>	! (+ - ; IDENT NUMfloat NUMint float for if int print scan while {	! (+ - ; IDENT NUMfloat NUMint else float for if int print scan while { }
<declaration>	float int	! (+ - ; IDENT NUMfloat NUMint else float for if int print scan while { }
<identList>	IDENT	;
<restoIdentList>	λ ,	;
<forStmt>	for	! (+ - ; IDENT NUMfloat NUMint else float for if int print scan while { }
<optExpr>	! λ (+ - IDENT NUMfloat NUMint	) ;
<ioStmt>	print scan	! (+ - ; IDENT NUMfloat NUMint else float for if int print scan while { }
<outList>	IDENT NUMfloat NUMint STR	)
<out>	IDENT NUMfloat NUMint STR	) ,
<restoOutList>	λ ,	)
<whileStmt>	while	! (+ - ; IDENT NUMfloat NUMint else float for if int print scan while { }
<ifStmt>	if	! (+ - ; IDENT NUMfloat NUMint else float for if int print scan while { }
<elsePart>	λ else	! (+ - ; IDENT NUMfloat NUMint else float for if int print scan while { }
<expr>	! (+ - IDENT NUMfloat NUMint	) ;
<atrib>	! (+ - IDENT NUMfloat NUMint	) ;
<restoAtrib>	λ =	) ;
<or>	! (+ - IDENT NUMfloat NUMint	) ; =
<restoOr>	λ ||	) ; =
<and>	! (+ - IDENT NUMfloat NUMint	) ; = ||
<restoAnd>	&& λ	) ; = ||
<not>	! (+ - IDENT NUMfloat NUMint	&&) ; = ||
<rel>	(+ - IDENT NUMfloat NUMint	&&) ; = ||
<restoRel>	!= λ < <= == > >=	&&) ; = ||
<add>	(+ - IDENT NUMfloat NUMint	!= &&) ; < <= = == > >= ||
<restoAdd>	λ + -	!= &&) ; < <= = == > >= ||
<mult>	(+ - IDENT NUMfloat NUMint	!= &&) + - ; < <= = == > >= ||
<restoMult>	% λ * /	!= &&) + - ; < <= = == > >= ||
<uno>	(+ - IDENT NUMfloat NUMint	!= % &&) * + - / ; < <= = == > >= ||
<fator>	(IDENT NUMfloat NUMint	!= % &&) * + - / ; < <= = == > >= ||

[Cuidado] Foi encontrada uma intercesão entre o conjunto first e follow da variável <elsePart>. Terminais: ['else']

Fonte: Autor

6 CONSIDERAÇÕES FINAIS

6.1 Conclusões

De acordo com a proposta inicial do trabalho, a ferramenta desenvolvida obteve sucesso nos testes de validação e apresentou bons resultados. Vista a necessidade da corretude ao trabalhar com o ambiente acadêmico, neste ponto o *software* desenvolvido mostrou-se útil. Apesar de não obter os resultados esperados em relação a interface com o usuário, a saída dos dados se mostrou satisfatória, apresentando o caminho a ser realizado nas manipulações gramaticais disponíveis pela ferramenta. Outro ponto importante foi a validação dos algoritmos disponibilizados pela literatura. Todos os pseudocódigos utilizados se mostraram corretos, ao aplicá-los em um ambiente real. Por fim, o *software* GTK tem como propósito o auxílio do aluno em sala de aula, para manipular de forma automática as gramáticas que serão utilizadas para a implementação de compiladores.

6.2 Trabalhos Futuros

Mediante os resultados obtidos e a proposta de uma abordagem didática para a construção de gramáticas para compiladores, como proposta de trabalho futuro seria interessante dar continuidade elaborando uma interface gráfica mais amigável ao usuário. Visto que nem todo aluno possui afinidade com a interface por linha de comandos, a confecção de uma interface mais intuitiva, podendo ser manipulada via cliques de *mouse*, seria uma boa proposta de trabalho futuro. Além do discorrido acima, um outro melhoramento poderia ser realizado. Visto que o GTK provê funções de auxílio na construção de gramáticas, o mesmo poderia também poderia gerar automaticamente analisadores léxico e sintáticos com as gramáticas já construídas do usuário. Desta forma, o *Grammar Tool Kit* seria um *software* de grande ajuda para a implementação de todo o *front-end* de um compilador.

7 Referências

VIEIRA, Newton José. **Introdução aos Fundamentos da Computação**. 1ª Edição. São Paulo: Pioneira Thompson Learning, 2006. 319p. ISBN-85-221-0508-1.

AHO, Alfred V. et al. **Compiladores - Princípios, técnicas e ferramentas**. 1ª Edição, São Paulo : Pearson Addison-Wesley, 2008. 634p. ISBN-978-85-88639-24-9.

DELAMARO, Marcio. **Como Construir um Compilador Utilizando Ferramentas Java**. 1ª Edição, São Paulo : Novatec Editora Ltda, 2004, 308p. ISBN: 85-7522-055-1.

HOPCROFT, John E. et al. **Introduction to Automata Theory, Languages, and Computation**. 2nd Ed, Addison Wesley, 2001, 521p. ISBN: 0-201-44124-1.

PARR, Terence. **ANTLR-Another Tool For Language Recognition**. 2008. Disponível em: www.antlr.org. Acesso em: 10 out. 2018.

Hacking Off: A Project Repository. 2012. Disponível em: hackingoff.com. Acesso em: 10 out. 2018.